Sieci Neuronowe

Notatki do wykładu „Sieci Neuronowe” dla studentów kierunku Informatyka
na Uniwersytecie Wrocławskim

Anna Bartkowiak

2 marca 2002

Anna Bartkowiak
Uniwersytet Wrocławski
Instytut Informatyki
Zakład Metod Numerycznych
Przesmyckiego 20
51–151 Wrocław
http://www.ii.uni.wroc.pl/~aba/

Adres elektronicznej wersji notatek:
http://www.student.ii.uni.wroc.pl/~aNeutrin/sieci/ksiazka/

Skład w LaTeX 2ε: autor i studenci
Korekta: autor i studenci

Niniejsza monografia jest zapisem semestralnego wykładu z Sieci Neuronowych *prowadzonego przez prof. Annę Bartkowiak w Instytucie Informatyki na Uniwersytecie Wrocławskim w latach 2001/2002.*

Zebrane tu materiały pojawiały się stopniowo — każdy rozdział odpowiada kolejnemu wykładowi — będąc jednocześnie w sposób praktyczny wykorzystywane przez studentów na zajęciach w laboratoriach komputerowych.

Spis treści

Podstawowe Oznaczenia

Wszystkie wektory — jeżeli tego nie zaznaczono wyraźnie — są wektorami kolumnowymi

d	–	liczba cech, składowych wektora danych (u statystyków oznaczana symbolem p, w MatLabie: R, u Osowskiego N)
H lub K	–	liczba neuronów warstwy ukrytej
P lub N	–	liczba wektorów uczących (liczebność próbki), od *pattern* (u statystyków: rozmiar próbki n lub N)
M lub S	–	liczba wyjść sieci neuronowej, czyli liczba neuronów ostatniej warstwy sieci, w MatLabie S
$\mathbf{x} = (x_1, \ldots, x_d)^T$	–	wejściowy wektor danych (wektor kolumnowy) (u statystyków: osobnik, jest pisany jako wektor wierszowy), często pisze się górne wskaźniki bez nawiasów okrągłych
$\ddot{\mathbf{x}} = (1, x_1, \ldots, x_d)^T$	–	wejściowy wektor danych w którym jako pierwszy element podstawiono tożsamościowo jedynkę (wtedy wektor wag otrzymuje jako pierwszą składową wartość w_0 oznaczającą stałą b_0 (bias) neuronu
$\mathbf{x}^{(n)} = (x_1^{(n)}, \ldots, x_d^{(n)})^T$	–	wejściowy wektor danych (wektor kolumnowy) o numerze n (u statystyków: n-ty osobnik, pisany jako wektor wierszowy); w MatLabie oznaczany jako wektor $\mathbf{p}$
$\ddot{\mathbf{x}}^{(n)} = (1, x_1^{(n)}, \ldots, x_d^{(n)})^T$	–	wektor danych dla p-tego wzorca, powiększony o element „1” stojący na pierwszym miejscu
$\mathbf{y} = (y_1, \ldots y_M)^T$	–	wyjściowy wektor wyników, wyprodukowanych przez sieć na podstawie wektora wejściowego $\mathbf{x}$. Pisząc bardziej dokładnie, mamy $\mathbf{y} = \mathbf{y}(\mathbf{x})$.

$\mathbf{y}^{(n)} = (y_1^{(n)}, \dots y_M^{(n)})^T$ – to samo, co poprzednio, ale napisane explicite dla n-tego osobnika.
Mamy oczywiście: $\mathbf{y}^{(n)} = \mathbf{y}(\mathbf{x}^{(n)})$

$\mathbf{t} = (t_1, \dots t_M)^T$ – docelowy wektor wyników (target). Sieć – na podstawie wektora wejściowego $\mathbf{x}$ ma wyprodukować wynik $\mathbf{y} = (y_1, \dots, y_M)^T$ możliwie zbliżony do wartości wektora docelowego $\mathbf{t} = (t_1, \dots t_M)^T$; w przypadku sieci z jednym wyjściem wektor wartości docelowych redukuje się do skalaru

$\mathbf{t}^n = (t_1^n, \dots t_M^n)^T$ – to samo, co poprzednio, ale napisane explicite dla n-tego osobnika, czyli dla n-tego wzorca

w_{ij} – waga z jaką i-ty neuron sumuje j-tą cechę, lub aktywację j-ego neuronu poprzedniej warstwy, $i = 1, \dots, H, \quad j = 0, 1, \dots, d$

$\mathbf{w}_i = (w_{i1}, w_{i2}, \cdots, w_{ij})^T$ – wektor wag i-tego neuronu, przy założeniu, że i-ty neuron otrzymuje sygnały od j innych neuronów; w zależności od kontekstu może to być również wektor zawierający na pierwszym miejscu stałą $w_{i0} = b_{i0}$, czyli *bias* i-tego neuronu: $\mathbf{w}_i = (w_{i0}, w_{i1}, w_{i2}, \cdots, w_{ij})^T$

$w_{ji}^{(k)}$ – oznacza wagę połączenia pomiędzy j - tym neuronem warstwy k-tej oraz i-tym neuronem warstwy $k-1$-szej

$\mathbf{W}_{H\times(d+1)}$ – tablica wag, w której wiersze i oznaczają numery neuronów w rozważanej warstwie (zakładamy tu, że warstwa ta zawiera H neuronów)
kolumna zerowa zawiera tzw. wagi zerowe (obciążenia, bias)
kolumny $1, \dots, d$ oznaczają numery cech w danych wejściowych

$$\mathbf{W}_{H\times(d+1)} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_H^T \end{bmatrix}$$

$\mathbf{W}_{H\times d}, \quad \mathbf{b}_{H\times 1}$ – tablica wag o znaczeniu podobnym, jak w poprzednim punkcie, jednak bez wag zerowych, które wyodrębniono w postaci wektora kolumnowego $\mathbf{b}$

u_i lub v_i – wynik sumowania bodźców dochodzących do neuronu nr i od neuronów (wejść) poprzedniej warstwy; kombinacja liniowa obliczana przez i-ty neuron na podstawie wczytanego wektora danych $\mathbf{x}$ wynosi odpowiednio:

$$u_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} \quad \text{lub} \quad u_i = \sum_{j=1}^{d} w_{ij} x_j + b_i$$

W MatLabie wynik u_i jest oznaczany jako n_i. Przyjmując tożsamościowo $x_0 \equiv 1$ oraz $b_i \equiv w_{i0}$, wynik u_i możemy zapisać prościej jako

$$u_i = \sum_{j=0}^{d} w_{ij} x_j = \mathbf{w}_i^T \mathbf{x}$$

$f(\cdot)$ – Funkcja aktywacji. Może być: skokowa (f. Heaviside´a, inaczej hardlimit), liniowa, sigmoidalna (logistyczna lub tangensoidalna), typu softmax.

Rozdział 1

Wstęp

1.1. Plan zajęć

1. Wprowadzenie w tematykę sieci neuronowych, trochę historii, neurony biologiczne i synapsy, określenie perceptronu. 'Neural Network Toolbox' MatLab'a, Obliczenia z MatLabem. 4+4 g.
2. Sieć jednokierunkowa jedno- i wielowarstwowa, funkcje aktywacji i znajdywanie wag, diagramy Hintona. 4+4 g.
3. Pakiet Netlab. Przykłady realizacji zagadnień regresyjnych lub aproksymacyjnych; porównanie z metodami statystycznymi. 4+4 g.
4. Sieci neuronowe o radialnych funkcjach bazowych, wybór liczby funkcji bazowych, stabilność rozwiązania. 4+4 g.
5. Przykłady zastosowań sieci neuronowych w klasyfikacji i diagnostyce. 4+4 g.
6. Samoorganizujące się sieci Kohonena, odwzorowanie punktów z wielowymiarowych przestrzeni z zachowaniem topologii konfiguracji, wizualizacja Sammona, pakiet SOM_PAK. 4+4 g.
7. Przykłady zastosowań sieci Kohonena do wizualizacji przebiegu procesu produkcyjnego, w marketingu usług turystycznych i ocenie stabilności banków. Inne typy sieci neuronowych i przykłady ich zastosowań.

Pracownia będzie oparta na pakiecie MatLab; będą wykorzystywane:
`MatLab, Neural Network Toolbox`,
`NETLAB` (autorzy: Ian Nabney i Ch. Bishop),
`http://www.ncrg.aston.ac.uk`
`SOM Toolbox` (autorzy: J. Vesanto i inn.)
`http://www.cis.hut.fi/projects/somtoolbox`

1.2. Linki do plików z danymi:

http://www.ics.uci.edu/pub/machine-learning-databases/
http://www.phys.uni.torun.pl/kmk/
http://citeseer.nj.nec.com/prechelt94proben/
http://kdd.ics.uci.edu/databases/tic/tic.html

O sieciach Kohonena:

http://www.student.ii.uni.wroc.pl/~Vol/

Rozdział 2

Od biologicznego neuronu do perceptronu i madaline

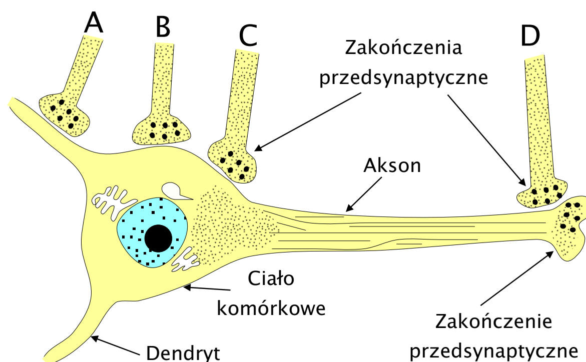
2.1. Elementy neurobiologii. Komórka nerwowa i jej struktura

Mózg i system nerwowy nie stanowią struktury ciągłej, ale składają się z około tryliona (10^{18}) komórek, z czego około 100 miliardów (10^{11}) stanowią komórki nerwowe połączone w sieci (Korbicz i inn. [8]), dzięki którym realizowane są funkcje inteligencji, emocji, pamięci i zdolności twórczych.

Ciało komórki nerwowej (inaczej: neuronu) jest dość podobne do komórek innych tkanek, wyróżnia się jednak wielkością otaczających ciało wypustek w postaci rozkrzewionych gałązek, tzw. dendrytów.

Z komórki nerwowej wychodzi długie włókno, nazywane **aksonem**, które na ogół rozgałęzia się w postaci tzw. drzewka aksonowego. Akson, rozwidlając się, dociera do wielu komórek, niemniej sygnał wyjściowy jest identyczny dla wszystkich odbiorców. Zakończenia gałązek aksonu stykają się z dendrytami innych neuronów, a miejsce styku nazywa się **synapsą**.

Podstawowe zadanie neuronu sprowadza się do przyjmowania (poprzez dendryty), przetwarzania i dalszego przekazywania (poprzez akson) informacji w postaci bodźców elektrycznych. W fizjologii pobudzanie aksonu określa się jako *wszystko albo nic*. Oznacza to, że dostatecznie silny bodziec powoduje każdorazowo tę samą reakcję, zbyt słaby bodziec nie wyzwala żadnej reakcji. Każdy nadchodzący synapsą bodziec dochodzi do ciała komórkowego.
Przewodzenie poprzez synapsy następuje zawsze tylko w jednym kierunku.
Informacja wzdłuż wypustek (aksonów, dendrytów) jest przenoszona w postaci impulsów elektrycznych, nazywanych potencjałami czynnościowymi.



Rysunek 2.1: *Schemat biologicznego neuronu na podstawie książki Michajlika i Ramotowskiego [5], str. 372, i pracy A. Kotuli [9]. Synapsy neuronu to:*
A – *synapsa aksonowo-dendrytowa,* **B** – *synapsa aksonowo-somatyczna,*
C – *synapsa aksonowo-aksonowa bliższa (zazwyczaj hamująca),*
D – *synapsa aksonowo-aksonowa dalsza (zawsze hamująca).*

Mózg[1], stanowiący centrum ludzkiego organizmu, jest częścią mózgowia obejmującą półkule mózgowe i część wzrokową podwzgórza. Mózgowie przeciętnie waży u człowieka około 1.3 kg. Mózg pokryty jest mocno pofałdowaną warstwą kory mózgowej. Warstwa ta ma grubość około 3 mm i powierzchnię 2500 cm^2. Kora mózgowa składa się z upakowanych gęsto komórek nerwowych o różnej wielkości i kształcie, tworzących kilka warstw. Liczbę neuronów szacuje się na 10 miliardów. Przyjmują one i wysyłają impulsy o częstotliwości $1 - 100\,Hz$, czasie trwania $1 - 2\,ms$, napięciu $100ms$ szybkości propagacji $1-100\,\frac{m}{s}$. Liczba połączeń między komórkami szacowana jest na 10^{15}. Tadeusiewicz w książce [15], str. 13, podaje, że szybkość pracy mózgu oszacować można na $10^{18}\ \frac{operacji}{s}$. Dla porównania[2], firma IBM planuje skonstruować w ciągu najbliższych pięciu lat komputer Blue Gene, który byłby w stanie wykonać 10^{15} operacji na sekundę. Komputer Blue Gene byłby w takim przypadku około dwa miliony razy potężniejszy niż dostępne dzisiaj komputery osobiste, tysiąc razy potężniejszy od komputera Deep Blue [3] i pięćset razy potężniejszy od najszybszych obecnie komputerów na świecie.

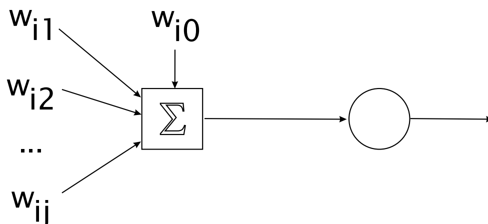
[1] na podstawie informacji podanych przez A. Kotulę [9]

[2] Kotula [9] — na podstawie informacji zawartych na stronach internetowych firmy IBM, dostępnych pod adresem `http://www.ibm.com`.

[3] Na podstawie stron internetowych firmy IBM, Deep Blue jest komputerem specjalnie zaprojektowanym do gry w szachy przez grupę specjalistów (Feng-Hsiung Hsu, Murray Campbell, Joe Hoane, Jerry Brody oraz C.J. Tan) pracujących dla firmy IBM. Projektowanie Deep Blue rozpoczęto w roku 1989, ale już od 1985 Hsu zajmował się tym problemem. W roku 1997 Deep Blue rozegrał pierwszy słynny mecz z najlepszym wówczas szachistą świata, Garrim Kasparowem, w meczu rewanżowym w maju 1997r wygrał Deep Blue — por. `http://www.research.ibm.com/deepblue/`.

2.2. Matematyczny model neuronu wg McCullocha i Pittsa

McCulloch i Pitts w roku 1943 jako pierwsi zaproponowali znacznie uproszczony w stosunku do rzeczywistego model neuronu, który do dziś jest podstawą większości modeli. Schemat takiego „matematycznego” działania neuronu jest podany na rysunku 2.2.



Rysunek 2.2: *Schemat działania neuronu o numerze i — według McCullocha i Pittsa*

Na rysunku tym mamy zaznaczony jeden neuron — ma on umownie numer i. Do neuronu tego zbiegają się sygnały (bodźce) — jest ich j. Neuron je sumuje — z wagami $w_{i1}, w_{i2}, \ldots, w_{ij}$ odpowiednio. Gdy obliczona wartość sumy przekroczy pewną wartość progową w_{i0}, specyficzną dla danego neuronu, następuje jego „zapłon”, inaczej mówiąc, neuron ten znajdzie się w stanie pobudzenia. Matematycznie stan pobudzenia neuronu wyraża się dwiema wartościami: 0, gdy pobudzenie neuronu nie przekroczyło jego specyficznej wartości progowej, i 1, gdy jest przeciwnie.
Spróbujmy teraz te fakty zapisać matematycznie. Będziemy rozpatrywać neuron o numerze i ze specyficzną wartością progową w_{i0}. Założymy, że stan pobudzenia neuronu jest zjawiskiem dyskretnym zmieniającym się w czasie τ w stałych odstępach czasu $\Delta\tau$. Oznaczmy wartość pobudzenia i-go neuronu w czasie τ symbolem $z_i(\tau)$. Oczywiście wartość neuronu w chwili $\tau + \Delta\tau$ zależy od tego, jak były pobudzone (dostarczające mu bodźce) neurony z jego otoczenia oznaczone tu umownie jako zbiór $\{j\}$ — w momencie poprzedzającym moment τ. Neuron oblicza sumę ważoną dostarczanych mu sygnałów. O ile po dodaniu do wyznaczonej przez neuron sumy wartości progowej otrzyma się liczbę dodatnią, następuje zapłon. Wyrazić to można w sposób następujący:

$$z_i(\tau + \Delta\tau) = \Theta(\sum_j w_{ij} z_j(\tau) + w_{i0}) \tag{2.1}$$

Zmienna $z_i(\tau)$ może mieć wartość 1, gdy i-ty neuron znajduje się w chwili τ w stanie zapłonu, lub 0, gdy tak nie jest.
Wagi w_{ij} występujące w powyższym wzorze odzwierciedlają istotność synapsy łączącej neuron i-ty i j-ty. Wagi mogą przyjmować zarówno dodatnie jak i ujemne

wartości:

$$w_{ij} \begin{cases} > 0 & : \text{ odpowiednik synapsy pobudzającej} \\ = 0 & : \text{ odpowiednik braku połączenia pomiędzy neuronami} \\ < 0 & : \text{ odpowiednik synapsy hamującej.} \end{cases}$$

Natomiast funkcja $\Theta(a)$ występująca we wzorze McCulocha i Pittsa to funkcja Heaviside'a (*hardlimit*) określona następująco:

$$\Theta(a) = \begin{cases} 1 & \text{dla} \quad a \geqslant 0 \\ 0 & \text{dla} \quad a < 0 \end{cases}$$

McCulloch i Pitts wykazali, że przy odpowiednio dobranych wagach w_{ij} synchroniczny zespół takich neuronów może wykonać te same obliczenia, co uniwersalna maszyna licząca.

Dalsze prace poszły w kierunku:

- użycia innych funkcji aktywacji — umożliwia to modelowanie procesów nieliniowych,
- przedstawienia sygnału z_i nie jako procesu dyskretnego, ale jako procesu ciągłego.

Stosuje się następujące uogólnienie modelu McCullocha i Pittsa:

$$z_i = g(\sum_j w_{ij} z_j + w_{i0}). \tag{2.2}$$

We wzorze tym nie uzależnia się stanu pobudzenia neuronu od czasu τ. Funkcja progowa $\Theta(\cdot)$ zastąpiona jest przez funkcję $g(\cdot)$, zwaną **funkcją aktywacji** (funkcją wygładzającą, funkcją przejścia, funkcją wzmocnienia). Model (2.2) uwzględnia aktualizację z_i w dowolnej chwili, umożliwia nieliniowość, z_i jest funkcją ciągła określającą stan neuronu w chwili t.

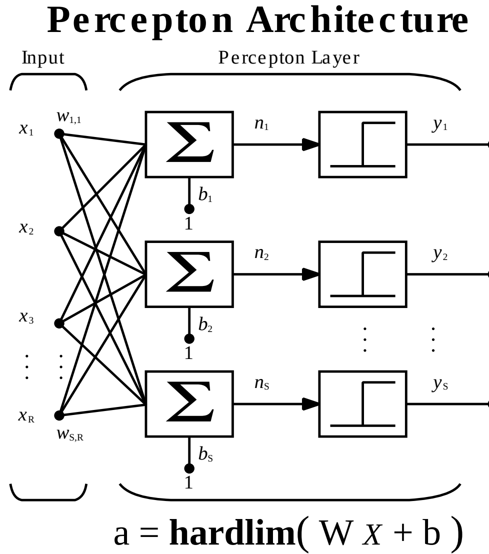
2.3. Perceptrony

Co to jest perceptron

Pojęcie perceptronu[4] zostało wprowadzone przez Rosenblatta[5] i przez długi czas było bardzo popularne, przede wszystkim w zagadnieniach związanych z rozpoznawaniem wzorców, lub — używając języka innego środowiska — zagadnień dyskryminacji i klasyfikacji danych.

[4]u Osowskiego perceptron nie jest specjalnie omawiany, chociaż na str. 22 mówi się o uczeniu bezgradientowym na zasadzie *reguły perceptronu*; pojęcie perceptronu jest omawiane specjalnie np. w książce Hertza i wsp., Rozdział 5, lub Korbicza i wsp., Rozdział 2

[5]Rosenblatt F., 'Principles of Neurodynamics', Washington D.C., Spartan Press 1962, również ten sam autor, 'The perceptron: A probabilistic model for information storage and organisation in the brain', Psychological Review, Vol. 65, 1958, str. 386-408



The Perceptron Learning Rule

$$W^{new} = W^{old} + ex^T$$

$$b^{new} = b^{old} + e$$

where

$$e = t - a$$

Rysunek 2.3: *Perceptron z R wejściami i S wyjściami*

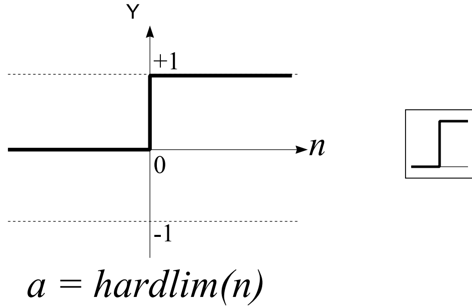
Rozpatrujemy sieć neuronową o następującej architekturze: R wejść, S neuronów w warstwie ukrytej, funkcja aktywacji typu hard_limit (Heaviside´a), liczba wyjść równa liczbie neuronów warstwy ukrytej (=K). Na wyjściu mogą pojawiać się tylko zera lub jedynki. Schemat takiej sieci jest przedstawiony na rysunku 2.3.

Funkcja aktywacji typu *hardlimit* jest przedstawiona na rysunku 2.4.

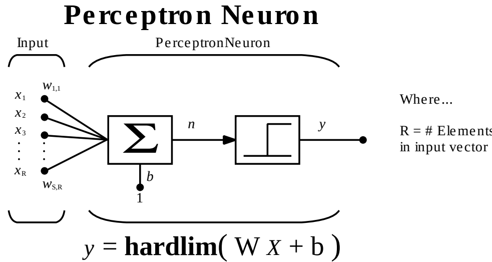
Idea perceptronu jest zawarta w następujących zasadach (por. Korbicz i wsp. [8]):

1. Perceptron jest siecią zbudowaną ze sztucznych neuronów; każdy sztuczny neuron — którego budowa jest przedstawiona na rysunku 2.5 — jest aktywowany funkcją *hardlimit* typu unipolarnego (zwracającą wartości 1 lub 0) lub bipolarnego (zwracającą tylko wartości 1 lub −1).

2. Sieć perceptronową można podzielić jednoznacznie na ściśle uporządkowane i rozłączne klasy elementów, zwanych warstwami, wśród których wyróżnić można *warstwę wejściową* i *wyjściową*. Pozostałe noszą nazwę *warstw ukrytych*.



Rysunek 2.4: *Funkcja aktywacji typu hardlimit, unipolarna*



Rysunek 2.5: *Perceptron z R wejściami, 1 neuronem warstwy ukrytej i 1 wyjściem*

3. Perceptron nie zawiera połączeń między elementami należącymi do tej samej warstwy.

4. Połączenia pomiędzy warstwami są asymetryczne i skierowane zgodnie z ich uporządkowaniem, tzn. od warstwy wejściowej do pierwszej warstwy ukrytej, następnie od pierwszej do drugiej warstwy ukrytej, itd. aż do warstwy wyjściowej. Nie ma połączeń zwrotnych.

Konsekwencją tych założeń jest, że każdy neuron — niezależnie od pozostałych — uczy się rozpoznawać pewną strukturę.

Popatrzmy na neuron o numerze s, $(s = 1, \ldots, S)$. Neuron ten najpierw oblicza sumę $u_s = b_s + \sum_{i=1}^{R} w_{si} x_i$, która następnie zostaje przetransformowana przez funkcję aktywacji typu hardlimit (por. rys. 2.4). Wynik transformacji, czyli wartość $g(u_s)$, zostaje skierowana bezpośrednio na wyjście.

Stała b_s jest tzw. zerowym obciążeniem (*bias*).

Nie ma dodatkowego przetwarzania przez neuron warstwy wyjściowej, czyli na wyjściu pojawiają się bezpośrednio wartości $g(u_s)$ — przyjmujące wartości 1 lub 0, odpowiednio.

Tak zdefiniowana sieć — perceptron nadaje się znakomicie do zagadnień klasyfikacyjnych, gdy naszym celem jest zaklasyfikowanie wektora danych $[x_1, \ldots, x_R]$ do jednej z S klas danych. Działanie perceptronu można w tym przypadku interpretować następująco:

Każdy neuron „specjalizuje" się w rozpoznawaniu jednej, np. s-tej klasy danych.

W trakcie uczenia neuron przystosowuje swoje wagi tak, aby móc rozpoznać próbki (tj. wektory danych) należące do s-tej klasy. „Obce" próbki (tzn. nie należące do s-tej klasy) są przez ten neuron odrzucane.

Wtedy, na zakończenie procesu uczenia, wektor wagowy $\mathbf{w}_s = [w_{s1}, \ldots, w_{sR}]$ wraz ze stałą b_s tworzy kombinację liniową zmiennych wejściowych $x_1, \ldots, x_R$ czyli wartość $u_s = b_s + \sum_{i=1}^{R} w_{si}x_i = b_s + \mathbf{w}_s^T\mathbf{x}$, taką że;

- duże wartości u_s świadczą za tym, że przedstawiany wektor $\mathbf{x}$ jest podobny do elementów s-tej klasy;
- małe wartości u_s świadczą za tym, że przedstawiany wektor $\mathbf{x}$ jest niepodobny do elementów s-tej klasy;

(Własności te wynikają z własności funkcji *hardlimit*)

Perceptron tak zdefiniowany potrafi nauczyć się rozpoznawać próbki należące do różnych klas **pod warunkiem, że klasy te są liniowo separabilne,** czyli rozłączne.

Proces uczenia („reguła perceptronu") jest bardzo prosta. Każdy neuron (s) powinien znaleźć taki wektor wag $\mathbf{w}_s$ i stałą b_s, które by pozwoliły na efektywne rozpoznanie struktury danych należącej do klasy s. Proces znajdywania odpowiednich wag odbywa się bezgradientową metodą iteracyjną omówioną w następnym podrozdziale.

Rozenblatt pokazał, że jeśli klasy są liniowo separabilne, to zaproponowany przez niego algorytm potrafi znaleźć wektory wagowe pozwalające na poprawne rozpoznanie elementów należących do poszczególnych klas.

2.4. Perceptron prosty — Klasyfikacja do dwóch klas

W przypadku dwóch klas wystarczy jeden neuron warstwy ukrytej, który będzie się specjalizował w rozpoznawaniu próbek pochodzących z klasy pierwszej. Schemat takiej sieci jest pokazany na rysunku 2.5. Oblicza się wtedy jedną wartość $u = b_0 + w_1x_1 + \cdots + w_Rx_R$. Duże wartości u świadczą za przynależnością próbki do klasy pierwszej, a małe wartości za przynależnością do klasy drugiej. Granicą rozdzielającą obie klasy jest hiperpłaszczyzna L

$$L: \quad b_0 + w_1x_1 + \cdots + w_Rx_R = 0. \tag{2.3}$$

Zauważmy, że wektor **w** tworzący hiperpłaszczyznę (2.3) jest wektorem ortogonalnym do tej hiperpłaszczyzny leżącym w półprzestrzeni w której powinna znajdować się rozpoznawana klasa.

W przypadku tylko dwóch zmiennych wejściowych x_1 i x_2 (czyli: $R = 2$) hiperpłaszczyzna (2.3) redukuje się do prostej $b_0 + w_1 x_1 + w_2 x_2 = 0$. Prosta ta rozdziela płaszczyznę (x_1, x_2) na dwie części: tam, gdzie $y > 0$ oraz gdzie $y <= 0$. Wartości dodatnie świadczą o przynależności próbki do klasy 1, A wartości zerowe lub ujemne wskazują na przynależność do klasy 2. Sytuacja taka jest pokazana na rysunku 2.6.

Rysunek 2.6: *Perceptron określony dla 2 zmiennych x_1 i x_2. Granica decyzyjna (decision boundary) dla określania przynależności do jednej z dwóch klas danych jest określona prostą L o równaniu* **w x**$+b = 0$, *gdzie* $\mathbf{W} = [w_{1,1}\ w_{1,2}]$. *Prosta ta rozdziela elementy klasy 1 i klasy 2*

2.5. Uczenie (trenowanie) perceptronu — reguła perceptronu

Zasadę tę wyrażono w opisie `MatLab NN Toolbox` za pomocą frazy, że uczenie sieci jest dokonywane „by altering the vector **w** toward input vectors to be classified as 1 and away from vectors to be classified as 0". Zasada ta jest zilustrowana na rysunku 2.7.

Uczenie każdego neuronu przebiega niezależnie od pozostałych.

Sieć uczymy (trenujemy) na podstawie tzw. próbki uczącej. Celem jest wyznaczenie wag, na podstawie których można, dla danego wektora **x**, wyznaczyć wartości y możliwie zgodne z podanymi wartościami docelowymi t (Przypomnijmy, że perceptron wytwarza na wyjściu jako y tylko wartości 1 lub 0; natomiast wartości docelowe (t) podawane w próbce uczącej, powinny określać, czy chcemy otrzymać na danym wyjściu wartość 1 czy też 0, co można deklarować conajmniej na 2 sposoby).

Przypuśćmy, że sieć ma tylko jedno wyjście, czyli $M = 1$. Niech e oznacza błąd sieci:

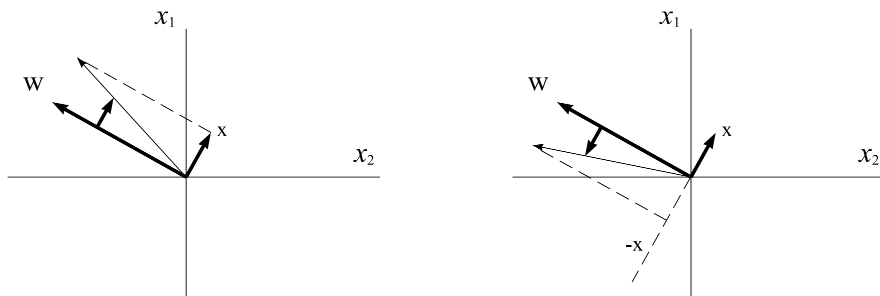
$$e = d - y.$$

Zauważmy, że błąd ten może przyjmować tylko trzy wartości: 0, 1 i -1.

Zasada uczenia (trenowania) sieci jest następująca:

Przyjmujemy (np. losowo) jakiś wektor **w** jako wektor wag, i jakąś początkową wartość obciążenia b.

Przedstawiamy sieci (nowy, lub też powtórnie, stary) wektor **x** dla którego jest znana wartość docelowa t. Obliczamy błąd $e = t - e$. W zależności od wielkości błędu obliczamy poprawkę $\Delta\mathbf{w}$ i Δb rozważając trzy przypadki:



Rysunek 2.7: *Poprawianie wag w trakcie uczenia. Lewa: Wektor* **x** *nie został wykazany jako należący do grupy 1 ($d - y = e = +1$). Wobec tego wektor* **w** *należy dosunąć do wektora* **x** *obliczając nowy wektor wag jako* $\mathbf{w}^{new} = \mathbf{w}^{old} + \mathbf{x}$ *Prawa: Wektor* **x** *został rozpoznany błędnie jako należący do grupy 1 ($d - y = e = -1$). Wobec tego wektor wag* **w** *należy odsunąć od wektora* **x** *odejmując ten ostatni od wektora* **w**. *Uwaga: Fakt, że wyrysowany wektor* **x** *jest* $\perp$ *do* **w** *— jest całkowicie przypadkowy. W rzeczywistości kąt między tymi dwoma wektorami może być zupełnie dowolny*

Przypadek 1. Obliczona wartość błędu jest równa zero, czyli $e = t - y = 0$. Oznacza to, że prognoza y była dobra, a wiec nic nie musimy zmieniać. Praktycznie mamy tu: $\Delta\mathbf{w} = 0$ i $\Delta b = 0$.

Przypadek 2. Obliczona wartość błędu jest równa 1, czyli $e = t - y = 1$. Oznacza to, że sieć prognozowała y błędnie. Powinna była zaprognozować „1” a zaprognozowała „0”. Można stąd wnioskować, że wagi nie są dostatecznie zaadaptowane do tego, żeby rozpoznawać wektory klasy „1”. Wobec tego robimy taką poprawkę, żeby zbliżyć wektor wag $\mathbf{w}$ do wektora $\mathbf{x}$.

Praktycznie: dodajemy wektor $\mathbf{x}$ do wektora $\mathbf{w}$, co można zapisać w postaci reguł: $\Delta\mathbf{w} = \mathbf{x}$ i $\Delta b = 1$.

Przypadek 3. Obliczona wartość błędu jest równa -1: $e = t - y = -1$. Oznacza to, że sieć prognozowała y błędnie. Powinna była prognozować „0” a prognozowała „1”. Można stąd wnioskować, że wagi zbyt często prognozują „1”, nawet gdy mamy do czynienia z wektorami klasy przeciwnej. W tej sytuacji robimy poprawkę mającą na celu odsunięcie wektor wag $\mathbf{w}$ od aktualnie przedstawianego sieci wektora $\mathbf{x}$.

Praktycznie: odejmujemy wektor $\mathbf{x}$ od wektora $\mathbf{w}$, co można zapisać w postaci reguł: $\Delta\mathbf{w} = -\mathbf{x}$ i $\Delta b = -1$.

Nasze postępowanie jest zilustrowane na rysunku 2.7, gdzie pokazano dodawanie i odejmowanie wektora $\mathbf{x}$ do (od) wektora $\mathbf{w}$. Wymienione trzy przypadki można zapisać razem w postaci jednego wyrażenia:

$$\Delta\mathbf{w} = (t - y)\mathbf{x}^T = e\mathbf{x}^T, \text{ oraz } \Delta b = (t - a)(1) = e,$$

lub

$$\mathbf{w}^{new} = \mathbf{w}^{old} + e\mathbf{x}, \quad b^{new} = b^{old} + e. \tag{2.4}$$

We wzorze powyższym symbol $\mathbf{w}$ oznacza wektor wagowy dowolnego neuronu. Zarówno $\mathbf{w}$ jak i $\mathbf{x}$ są wektorami kolumnowymi.

Jeżeli mamy sieć składającą się z S neuronów w warstwie ukrytej, to wszystkie wektory wagowe możemy zestawić w macierz $\mathbf{W}$ o wymiarach $S \times R$:

$$\mathbf{W}_{S\times R} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_S^T \end{bmatrix} \tag{2.5}$$

Wtedy regułę uczenia całego perceptronu (czyli wszystkich jego wektorów wagowych) możemy zapisać za pomocą jednego równania:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{x}^T,$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e},$$

gdzie $\mathbf{W} = \mathbf{W}_{S\times R}$, $\quad \mathbf{e} = (e_1, \ldots, e_S)^T$, no i oczywiście $\mathbf{x} = (x_1, \ldots, x_R)^T$.

2.6. Kiedy model perceptronowy daje dobre wyniki

Przedstawiona metoda (dyskryminacji liniowej) jest względnie prosta i jest dość szeroko używana w pakietach komercyjnych przeznaczonych do analizy danych pomiarowych.

Należy jednak zaznaczyć, że metoda ta działa dobrze, gdy klasy są rozłączne, czyli mają własność liniowej separabilności *linear separability.*

Wtedy rzeczywiście wystarczy prosta lub hiperpłaszczyzna żeby rozdzielić poszczególne klasy, a sieć neuronowa jest zdolna nauczyć się w skończonej liczbie kroków skonstruować hiperpłaszczyznę rozdzielającą (czyli wyznaczyć wektor wag, na podstawie którego taką hiperpłaszczyznę można określić).

Zasada perceptronu została dość zdecydowanie i pesymistycznie zaopiniowana w książce M. Minsky i S. Pappert, 'Perceptrons', Cambridge Ma, MIT Press, 1969. Pokazali oni, że perceptrony Rosenblatta mają ograniczoną zdolność uczenia się, a niektórych struktur nie potrafią się wręcz nauczyć. Opinia ta wpłynęła na zahamowanie — na okres conajmniej 10 lat — badań naukowych związanych ze sztucznymi sieciami neuronowymi.

Jednak jest znanych wiele przypadków, gdy analizowane dane nie posiadają własności „liniowej separabilności", i nie są to bynajmniej dane wydumane, ale występujące w praktyce.

W przypadku takich danych metoda perceptronową, tak jak ją przedstawiono wyżej, nie będzie dawać dobrych wyników i należy sięgnąć po inne metod, np. skorzystać ze sztucznej sieci neuronowej o wzbogaconej strukturze (np. sieci zawierające więcej neuronów lub wielowarstwowe). Innymi, bardziej efektywnymi metodami mogą również okazać się metody statystyczne, np.:

- Liniowa funkcja dyskryminacyjne Fishera
- Kanoniczne funkcje dyskryminacyjne
- Postępowanie oparte na ilorazie wiarogodności
- Modelowanie nieparametryczne (kernel density)
- Modele skomponowane z mieszanin rozkładów (mixture models)

2.7. Moduły MatLaba demonstrujące działanie perceptronu

MatLabowski Pakiet NN Toolbox oferuje następujące moduły demonstracyjne obrazujące działanie perceptronu:

1. `nnd4db`, Decision Boundaries
2. `nnd4pr`, Perceptron learning rule. Pick boundaries

3. demop1, Classification with a 2-input perceptron
4. demop4, Outlier input vector
5. demop5, Normalized perceptron rule
6. demop6, Linearly non-separable vectors.

Wszystkie moduły ilustrują działanie klasycznego perceptronu dla przypadku, gdy liczba cech (wymiar wektora wejściowego) wynosi 2. (W MatLab-ie wektor wejściowy jest oznaczany jako **p**, a jego składowe jako p(1) i p(2) — zamiast naszych oznaczeń x_1 i x_2; wartości docelowe są oznaczane jako t — od ang. słowa *target*).

W przypadku dwu-wymiarowych danych wejściowych granicę decyzyjną (czyli prostą $y = w_1x_1 + w_2x_2 + b$) można zilustrować na płaszczyźnie.

Moduły nnd4db i nnd4pr demonstrują uczenie się perceptronu — który w zależności od podanych danych wyznacza odpowiednie wagi w_1, w_2 oraz obciążenie (*bias*) b. Moduły te pozwalają na interakcyjne określanie danych wejściowych oraz manualne korygowanie prostej decyzyjnej. Moduły te pochodzą z książki Hagana i wsp. [3].

Na uwagę zasługuje moduł demop4: pokazuje on, jak mocno odstająca obserwacja może utrudnić uczenie się sieci.

2.8. Adaline i Madaline

Mniej więcej w tym samym czasie co Rosenblatt (to jest około r. 1960), Widrow wraz z współpracownikami zbudował dość podobną sieć co perceptron, ale opartą na liniowej funkcji adaptacji, która dopuszczała na wyjściu dowolne liczby rzeczywiste. Zbudowana przez Widrowa sztuczna sieć składała się z elementów nazywanych *Adaline*, czyli Adaptive Linear Element. Cała sieć jako wielokrotność (multiple) tych liniowych elementów „adaline” została nazwana *Madaline*.

Do tak zbudowanej sieci Widrow z współpracownikami określili regułę uczenia opartą na minimalizacji sumy kwadratów błędów, czyli metodzie najmniejszych kwadratów.

2.9. Typologia sieci ze względu na pewne jej własności formalne

2.9.1. Podstawowe architektury sieci neuronowych

Rozróżniamy następujące typy sieci neuronowych[6]

[6]Wiadomości głównie na podstawie książki Osowskiego [12], str. 18

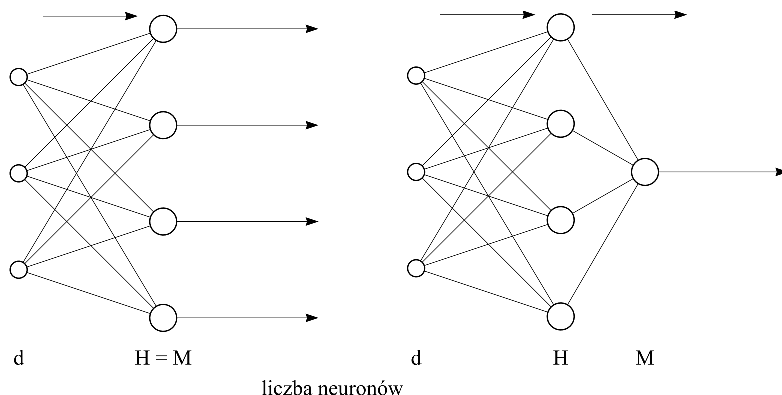
1. Sieci jednokierunkowe jednowarstwowe i wielowarstwowe.

   Warstwa wejściowa nie liczy się.

   Sieci jednowarstwowe używane są w praktyce bardzo rzadko (przykłady: perceptron prosty, adaline).

   Najczęściej używa się sieci dwuwarstwowych. Sieci takie składają się z warstwy wejściowej, warstwy ukrytej o H neuronach, i warstwy wyjściowej o M neuronach.

   Przykłady sieci jednokierunkowych jedno- i dwuwarstwowych są pokazane na rysunku 2.8.



Rysunek 2.8: *Przykłady sieci jednokierunkowych jedno- i dwuwarstwowych*

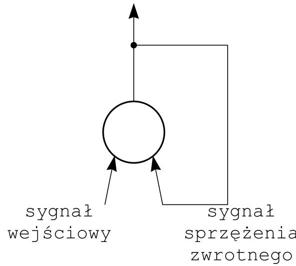
   Sieć dwuwarstwowa potrafi rozpoznać klasy obiektów, które geometrycznie są ulokowane w obszarach będących kombinacją iloczynów i sum mnogościowych stanów aktywnych. Przykładowo, dla perceptronu stan aktywny neuronu wyznacza półprzestrzeń; dla kilku neuronów te półprzestrzenie przecinają się i tworzą zbiory wielościenne wypukłe (por. Korbicz i inn. [8], str. 41, Tadeusiewicz [16] str. 107-108).

   Sieci o więcej niż 2 warstwach używa się przy modelowaniu skomplikowanych i wybitnie nieliniowych związków między danymi wejściowymi i wartościami docelowymi (targetem).

2. Sieci rekurencyjne, mogą być jedno- i wielowarstwowe, posiadają np. zdolność uwzględniania elementów opóźnienia przy analizie sygnałów. Znajdują szerokie zastosowanie w teorii sterowania.

3. Sieci komórkowe[7]. Neurony znajdują się w specyficznej konfiguracji geometrycz-

[7] Korbicz i inn. [8], str. 30; termin ten nie występuje u Osowskiego



Rysunek 2.9: *Przykład sprzężenia zwrotnego zachodzącego dla sygnałów przetwarzanych przez jeden neuron — na postawie książki Tadeusiewicza [16], str. 244*

nej umożliwiającej określenie sąsiedztwa. Występują sprzężenia zwrotne między elementami przetwarzającymi, ale tylko między elementami znajdującymi się w „sąsiedztwie".

2.9.2. Metody uczenia sieci

Wyróżniamy następujące metody uczenia:[8]

1. Uczenie pod nadzorem (z nauczycielem).

   Sieć najpierw — na podstawie próbki uczącej — uczy się rozpoznawać pewne klasy lub relacje. Sieć „uczy" się w ten sposób, że w miarę prezentacji wzorców jej wektory wagowe zostają zaadaptowane (zmodyfikowane) z myślą, aby zminimalizować błąd predykcji, czyli różnicę między wartością y prognozowaną przez sieć, a odpowiednią wartością docelową t podaną w próbce uczącej.

   Próbki (wzorce) przedstawiane sieci przy uczeniu mają postać

   $$\mathbf{x}^n, \mathbf{t}^n, \quad n = 1, \ldots, N,$$

   gdzie

   wskaźnik n indeksuje (numeruje) wzorce próbki uczącej,

   liczba N jest liczebnością próbki uczącej,

   $\mathbf{x}^n = [x_1^n, x_2^n, \ldots, x_d^n]^T$ – jest n-tym wektorem danych,

   $\mathbf{t}^n = [t_1^n, \ldots, t_M^n]$ – jest wektorem wartości docelowych, odpowiadających n-tej próbce,

   M – oznacza liczbę wyjść sieci.

[8] Osowski [12], str. 20–32, ze zmianą w oznaczaniu numeracji wzorców: u nas n, u Osowskiego p; i wartości docelowych: u nas t, u Osowskiego d

2. Uczenie z krytykiem. Jest to odmiana uczenia pod nadzorem. Sieć nie otrzymuje informacji o wartościach docelowych t, ale jedynie informacje, czy podjęta przez system akcja (np. zmiana wartości wag) daje wyniki pozytywne czy też negatywne w sensie pożądanego zachowania systemu.

3. Uczenie samoorganizujące się typu Hebba. Na zasadzie analogii do obserwacji neurobiologicznych (doświadczenia Pawłowa z bodźcami) przyjmuje się, że:

   - przy jednoczesnym stanie pobudzenia obu neuronów waga powiązań między tymi neuronami wzrasta,
   - jeżeli dwa neurony nie są jednocześnie pobudzone, to waga połączeń między nimi maleje.

   Uczenie jest procesem iteracyjnym, zmiana wag w kolejnych cyklach uczących może być zapisana następująco (k oznacza numer iteracji lub cyklu uczącego):

   $$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}(k), \quad \text{gdzie } \Delta w_{ij}(k) = F(x_j, y_i)$$

   przy czym $F(x_j, y_i)$ jest funkcją stanu sygnału wejściowego x_j, zwanego presynaptycznym, oraz stanu wyjściowego y_i, zwanego postsynaptycznym.

   W klasycznym ujęciu Hebba jako funkcję F przyjmowano funkcję iloczynową obu sygnałów, a zmiana wag wyrażała się jako:

   $$\Delta w_{ij}(k) = \eta x_j(k) y_i(k),$$

   przy czym η oznacza tzw. współczynnik uczenia.

   Uczenie neuronu z zastosowaniem uczenia Hebba może odbywać się w trybie bez nauczyciela lub z nauczycielem. Znajduje zastosowanie przy nadzwyczaj ważnym zagadnieniu redukcji wymiarowości (kompresji) danych. Istotny wkład przy opracowywaniu algorytmów określających tzw. czynniki główne (składowe główne Hotellinga, transformata Karhunen-Loeve'go, ang. *Principal components*) wnieśli T. Sejnowski, E. Oja i T.D. Sanger.

4. Uczenie samoorganizujące się typu konkurencyjnego. Najbardziej znanym typem sieci wykorzystującej ten sposób uczenia jest tzw. sieć Kohonena (SOM, Self Organizing Map).

   Specyficzną charakterystyką tego typu uczenia jest, że przy prezentacji nowego wzorca tylko jeden neuron może zostać pobudzony, a pozostałe pozostają w stanie spoczynku. Z tego powodu ten typ uczenia jest również określany jako WTA (*Winner Takes All*). Uczenie przebiega w ten sposób, że grupa neuronów współzawodniczących otrzymuje te same sygnały wejściowe $\mathbf{x}^n$. i oblicza swoją aktywację $u_i = \mathbf{w}_i^T \mathbf{x}^n$. W wyniku porównania aktywacji u_i (wskaźnik i indeksuje tu neurony współzawodniczące) zwycięża ten neuron, którego wartość u_i jest największa. Neuron zwycięzca przyjmuje na swoim wyjściu stan 1 (i może np. zaadaptować swoje wagi), natomiast pozostałe (przegrywające) przyjmują stan 0, a ich wagi pozostają niezmienione.

2.9.3. Zdolności do uogólnienia

Chcielibyśmy, aby sztuczne sieci neuronowe rozpoznawały również obiekty, czy też klasy obiektów, których nie pokazaliśmy w próbce uczącej. Niektóre sieci (np. sieć Fahlmana oparta o zasadę korelacji kaskadowej) potrafią rozpoznać, że przedstawiany wzorzec nie należy do żadnej dotychczas przez tę sieć rozpoznanych klas.

Ocieramy się tu o zagadnienie „inteligencji” sztucznej sieci neuronowej. Do niedawna uważano, że zdolność uogólniania jest właściwa tylko umysłowi ludzkiemu.

Rozdział 3

Sieci neuronowe jednokierunkowe dwuwarstwowe

3.1. Określanie architektury sieci

Architekturę sieci jednokierunkowych określają następujące parametry: liczba neuronów warstwy wejściowej, liczba warstw ukrytych oraz liczba neuronów w każdej z tych warstw, liczba neuronów warstwy wyjściowej, funkcje aktywacji dla każdej warstwy ukrytej i warstwy wyjściowej.

Generalnie sieć może wykonywać dwie funkcje:

1. uczyć się, tj. adaptować swoje wagi przy zadanej swojej architekturze $\mathbf{\Theta}$,
2. wykonywać swoje właściwe zadanie, tj. na podstawie swojego algorytmu wykonywać predykcję, czyli obliczać wartości $y(\mathbf{x}; \mathbf{W}, \mathbf{\Theta})$

Czynność 1 czyli uczenie sieci może być wykonywane wieloma sposobami przy użyciu różnych algorytmów.

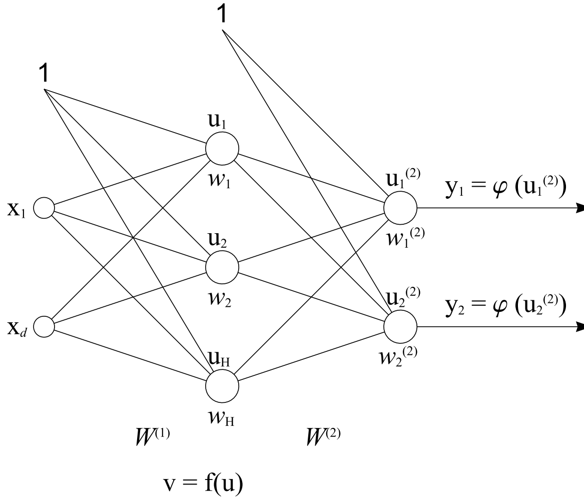
Czynność 2 jest już jednoznaczna.

Przykład sieci jednokierunkowej dwuwarstwowej jest podany na rysunku 3.1.

Architekturę tej sieci określają parametry $\mathbf{\Theta} = (d, H, M, f(.), \phi(.))$ o następującym znaczeniu:

d – liczba wejść, na rysunku: $d = 2$,

H – liczba neuronów warstwy ukrytej, na rysunku: $H = 3$,



Rysunek 3.1: *Sieć jednokierunkowa dwuwarstwowa. Neuron i ($i = 1, \ldots, H$) pierwszej warstwy oblicza sumę $u_i = u_i^{(1)} = \sum_{j=1}^{H} w_{ij} x_j$, a następnie swoją aktywację $v_i = f(u_i)$; podobnie neuron i z drugiej warstwy ($i = 1, \ldots, M$) oblicza ważoną sumę dochodzących do niego bodźców: $u_i^{(2)} = \sum_{j=1}^{M} w_{ij}^{(2)} v_j^{(1)}$, a następnie swoją aktywację $y_i = \varphi(u_i^{(2)})$*

M − liczba neuronów warstwy wyjściowej, na rysunku: $M = 2$,

$f(\cdot)$ oraz $\varphi(\cdot)$ − funkcje aktywacji warstwy ukrytej i warstwy wyjściowej

Neurony warstw ukrytych i warstwy wyjściowej spełniają rolę sumatorów dochodzących do nich — z warstw poprzednich — bodźców. Sumowanie jest ważone, w związku z tym każdy neuron ma przypisany swój odrębny wektor wagowy, z jakim sumuje dochodzące do niego bodźce.

W przypadku sieci dwuwarstwowej mamy dwa układy wektorów wagowych:

$\mathbf{w}_1, \ldots, \mathbf{w}_H$ – wektory wag przy neuronach warstwy ukrytej; $\mathbf{w}_h = [w_{h0}, w_{h1}, \ldots, w_{hd}]^T$, $h = 1, \ldots, H$ (wskaźnik h oznacza numer neuronu danej warstwy; drugi wskaźnik — numer neuronu poprzedniej warstwy),

$\mathbf{w}_1^{(2)}, \ldots, \mathbf{w}_M^{(2)}$ – wektory wag przy neuronach warstwy wyjściowej; $\mathbf{w}_j = [w_{j0}, w_{j1}, \ldots, w_{jH}]^T$, $j = 1, \ldots, M$,

Wektory te można umieścić w macierzach $\mathbf{W}^{(1)}$ i $\mathbf{W}^{(2)}$ z wyraźnym zaznaczeniem

— za pomocą górnego wskaźnika — której warstwy dane wagi dotyczą:

$$\mathbf{W}^{(1)}_{H\times(d+1)} = \begin{bmatrix} (\mathbf{w}_1^{(1)})^T \\ (\mathbf{w}_2^{(1)})^T \\ \vdots \\ (\mathbf{w}_H^{(1)})^T \end{bmatrix}, \qquad \mathbf{W}^{(2)}_{M\times(H+1)} = \begin{bmatrix} (\mathbf{w}_1^{(2)})^T \\ (\mathbf{w}_2^{(2)})^T \\ \vdots \\ (\mathbf{w}_M^{(2)})^T \end{bmatrix}.$$

Aby sieć funkcjonowała prawidłowo, tzn. wykonywała swoje zasadnicze czynności (mogą nimi być: predykcja, klasyfikacja, grupowanie na zasadzie podobieństwa lub niepodobieństwa, filtrowanie i odszumianie sygnałów), powinny być określone wektory wag występujące w algorytmie wykonywanym przez sieć. Wagi te na ogól są wynikiem uczenia (trenowania) sieci. (w rozdziale 2 omówiliśmy już trenowanie perceptronu, dzisiaj omówimy jeszcze trenowanie prostej sieci zawierającej elementy *adaline*).

3.2. Funkcjonowanie sieci po wytrenowaniu jej

Celem trenowania jest otrzymanie takich współczynników wagowych (wektorów wag), aby po przedstawieniu sieci jakiejś (opcjonalnie nowej) próbki czyli wektora $\mathbf{x}$ sieć mogła dokonać predykcji lub też rozpoznania, do jakiej kategorii wzorców dana próbka $\mathbf{x}$ należy.

Aby to zrobić, sieć wykonuje — dla każdego jej prezentowanego wektora $\mathbf{x}$ — następujące obliczenia przebiegające w dwóch etapach:

1. Najpierw każdy neuron warstwy ukrytej działa niezależnie jako sumator obliczając ważoną sumę sygnałów wejściowych:

   $$u_h = \mathbf{w}_h^T \mathbf{x}.$$

   Z obliczonej sumy u_h następuje wyznaczenie aktywacji neuronu v_h. Do wyznaczenia tej aktywacji używa się funkcji f zadanej przy określaniu architektury danej sieci:

   $$v_h = f(u_h), \quad h = 1, \ldots, H.$$

   Obliczone w punkcie 1 aktywacje $v_1, \ldots, v_H$ są przekazywane do następnej warstwy, w tym przypadku warstwy wyjściowej.

2. Przekazywane przez poprzednią warstwę wartości v_h stanowią dane dla drugiej warstwy (wyjściowej), której neurony działają podobnie jak neurony warstwy ukrytej. A mianowicie:

   Każdy neuron j, $(j = 1, \ldots, M)$ wykonuje najpierw funkcję sumatora, obliczając sumę

   $$u_j^{(2)} = (\mathbf{w}_j^{(2)})^T \mathbf{v}, \quad \text{gdzie } \mathbf{v} = [v_1, \ldots, v_H].$$

Obliczony wynik sumowania jest następnie przetwarzany przez funkcję aktywacji $\varphi(.)$ dostarczając ostatecznego wyniku y_j pojawiającego się na j-tym wyjściu sieci:

$$y_j = \varphi(u_j^{(2)}).$$

3.3. Uczenie *adaline* — reguła *delta* Widrowa-Hoffa

Adaline (Adaptive linear element) jest bardzo prostą siecią, podobną do perceptronu prostego. Różni się funkcją aktywacji (jest nią funkcja *signum* lub funkcja *purelin*, czyli czysto liniowa) i sposobem uczenia. Zasady funkcjonowania *adaline* zostały opracowane przez Widrowa i współpracowników w latach 60-tych. Proste sieci adaline połączone w sieć jednokierunkową tworzą sieć *Madaline*, tj. *Multiple adaline* — i sieci tego typu były używane do niedawna, a nawet podobno są używane jeszcze dzisiaj.

Omówimy teraz bardziej szczegółowo zasady uczenia sieci typu *adaline*[1], czyli sieci z jednym wyjściem.

Wprowadźmy następujące oznaczenia:

$\mathbf{x}^n = (x_{n1}, \ldots, x_{nd})^T$ – wektor danych (sygnał wejściowy),

t^n, y^n – znana wartość docelowa (target) i wartość wyprodukowana przez sieć jako odpowiedź na sygnał $\mathbf{x}^n$.

Różnica

$e_n = t^n - y^n$ stanowi tzw. błąd sieci.

$e_n^2 = (t^n - y^n)^2 = (t^n - y(\mathbf{x}^n; \mathbf{w}))^2$ stanowi kwadrat błędu.

Chcielibyśmy, aby ten kwadrat błędu był możliwie mały.

Jeżeli prezentacji było N $(n = 1, \ldots, N)$, to chcielibyśmy wyznaczyć wektor wagowy $\mathbf{w}$ w ten sposób, żeby

$$\sum_{n=1}^{N} e_n^2 = min!$$

Czyli faktycznie stosujemy metodę najmniejszych kwadratów (least squares error).

W zasadzie stosuje się dwie metody wyznaczania wag $\mathbf{w}$:

- Na zasadzie wyznaczenia wartości oczekiwanej kwadratu błędu $\mathsf{E}[e^2]$. Wielkość błędu jest uważana za zmienną losową, a wartość oczekiwana $\mathsf{E}[e^2]$ jest wyznaczana ze względu na rozkład prawdopodobieństwa zmiennych losowych $(\mathbf{x}, t)$.

  Jednak, aby obliczyć wartość oczekiwaną jakiejś zmiennej losowej, musimy znać rozkład prawdopodobieństwa tej zmiennej losowej.

  Metoda ta prowadzi do układu równań normalnych, których współczynnikami są kowariancje składowych wektora $(x_1, \ldots, x_d, t)$.

[1] Korbicz i inn., str. 47

Jeśli nie znamy rozkładu prawdopodobieństwa, a uważamy że posiadany zbiór wzorców jest reprezentatywny dla badanej populacji, to możemy przybliżyć wartość oczekiwaną sumy kwadratów błędów *zaobserwowaną* uśrednioną sumą kwadratów błędów, i szukać dla niej minimum. Również w tym przypadku dochodzimy do układu równań normalnych.

Bardziej dokładną metodą — w przypadku znajomości rozkładu prawdopodobieństwa błędu — jest metoda *największej wiarygodności*. W przypadku gdy rozkład błędu jest rozkładem normalnym, otrzymujemy ten sam układ równań normalnych.

- Na zasadzie iteracyjnego przybliżania wag.

  Prezentujemy sieci wiele razy (w różnej kolejności) wektory danych (nazywane również próbkami lub wzorcami).

  W każdym kroku k wagi ulegają adaptacji

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \Delta\mathbf{w},$$

  przy czym poprawka $\Delta\mathbf{w}$ może być wykonywana na wiele różnych sposobów.

  Możemy również prezentować sieci cały zbiór wzorców i szukać minimum schodząc coraz niżej — aż do minimum.

  Jedna iteracja (jeden krok iteracyjny) jest nazywany *epoką*.

W przypadku trenowania *adaline* Widrow i Hoff zastosowali następujące zasady:

Określili błąd z pominięciem aktywacji, i tak określony błąd oznaczyli symbolem δ. Tym samym, kwadrat błędu czyli funkcja δ^2 przy prezentacji n-tego wzorca jest określona jako:

$$\delta_n^2 = (t^n - \mathbf{w}^T\mathbf{x}^n)^2. \tag{3.1}$$

Uaktualnianie wektora wag $\mathbf{w}$ w k-tej iteracji, w której zaprezentowano sieci n-ty wzorzec $\mathbf{x}^n, t^n$ odbywa się wg już przedstawionej wyżej zasady

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \Delta\mathbf{w}, \tag{3.2}$$

przy czym jako $\Delta\mathbf{w}$ przyjmuje się najczęściej kierunek najszybszego spadku funkcji błędu, a to oznacza kierunek przeciwny do gradientu funkcji błędu:

$$\Delta\mathbf{w} = -\eta\nabla\delta_n^2(k). \tag{3.3}$$

Współczynnik η w powyższym równaniu oznacza tzw. współczynnik uczenia - zaraz powiemy o nim coś więcej, najpierw jednak zajmiemy się gradientem funkcji błędu.

Przy kwadracie błędu określonym wzorem (3.1) mamy

$$\nabla\delta_n^2 = \frac{\partial}{\partial\mathbf{w}}\delta_n^2 = -2\delta_n\mathbf{x}^n.$$

Stąd wynika wzór na poprawianie wektora **w** po kolejnej, k-tej iteracji:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + 2\eta\delta_n\mathbf{x}^n.$$

Reguła powyższa nosi nazwę **reguły Widrowa-Hoffa**; czasami jest nazywana **regułą delty** lub **regułą najmniejszego błędu średniokwadratowego**.

Wyznaczenie współczynnika η jest zadaniem skomplikowanym i wymaga znajomości statystyki prezentowanych wzorców wejściowych. Mówiąc ogólnie, współczynnik ten powinien być z zakresu:

$$0 < \eta < 1/\lambda_{max},$$

gdzie λ_{max} jest największą wartością własną macierzy $\mathbf{A} = E\{\mathbf{x}\mathbf{x}^T\}$. Jeżeli wzorce $\mathbf{x}$ są wystandaryzowane na (0,1), to macierz $\mathbf{A}$ jest macierzą korelacji między składowymi (cechami) danych wejściowych.

Ogólnie, jeśli η jest zbyt duże, to proces poszukiwania optymalnego wektora jest rozbieżny; z kolei, jeśli wartość η jest zbyt mała, to zbieżność jest powolna.

Nowsze algorytmy uzależniają wartość współczynnika η od czasu uczenia (numeru iteracji k) — w miarę upływu czasu uczenia η maleje.

Istnieją dwie modyfikacje reguły Widrowa-Hoffa[2]

Pierwsza polega na tym, że nie dokonuje się modyfikacji wag po każdej prezentacji wzorca, ale dopiero po prezentacji pewnej ich liczby, np. $N1$. Po każdej prezentacji jednego wzorca $\mathbf{x}^n$ oblicza się tylko poprawkę $\delta_n\mathbf{x}^n$; poprawki te się sumuje, a po prezentacji określonej liczby $N1$ wzorców uśrednia się je, a następnie — na podstawie takiej uśrednionej poprawki — uaktualnia się wektor wag. W rezultacie otrzymuje się dokładniejszą aproksymację gradientu błędu średniokwadratowego $\nabla\delta^2$.

Druga modyfikacja polega na stosowaniu wzoru:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta(1-\rho)\delta_n\mathbf{x}^n + \rho(\mathbf{w}(k) - \mathbf{w}(k-1)).$$

Wprowadzony tu parametr ρ ($\rho \in (0,1)$) pozwala na zaniedbywanie zbyt dużych fluktuacji wektora wag i spełnia rolę tzw. *momentum* znanego w teorii optymalizacji.

[2] Korbicz i inn. [8], str. 48

3.3.1. Odfiltrowywanie szumów za pomocą sieci zawierającej blok *Tapped Delay Line*

Na bazie struktury Adaline są zbudowane sieci wykonujące tzw. adaptacyjne filtrowanie (*Adaptive Filtering*).

W szczególności buduje się sieci zawierające tzw. *Tapped Delay Line*. Na wejściu jest wprowadzany sygnał jednowymiarowy x(t) który jest zapamiętywany dla kilku (np. R) ostatnich momentów czasowych. Neuron sumujący zbiera R ostatnich sygnałów, oblicza z nich ważoną kombinację liniową (wagami są wartości wag $w_1, \ldots, w_R$), sumę przetwarza liniowo dodając do nich obciążenie w_0, a wynik przekazuje na wyjście:

$$y(t) = purelin(\mathbf{w}\ \mathbf{x} + b) = \sum_{i=1}^{R} w_i x(t-i+1) + w_0.$$

3.3.2. Uczenie *Madaline*

Sieć *Madaline* składa się z pojedynczych bloków *adaline* połączonych równolegle. Najczęściej stosujemy tu uczenie iteracyjne. Prezentowana sieci próbka ma postać (wskaźnik n oznacza tu indeks prezentowanej próbki, a wektor $\mathbf{t}$ oznacza prawidłowe wartości, które sieć powinna aproksymować):

$$\mathbf{x}^n, \mathbf{t}^n, \quad \text{gdzie} \ \ \mathbf{x}^n = (x_{n1}, \ldots, x_{nd})^T, \quad \mathbf{t}^n = (t_{n1}, \ldots, t_{nM})^T.$$

W rezultacie działania sieci otrzymujemy na M wyjściach M wartości

$$y_1^n, \ldots, y_M^n,$$

które zostały obliczone przez sieć z wczytanego wektora $\mathbf{x}^n$ i aktualnych wag zapamiętanych w wektorze $\mathbf{w}$.

Ponieważ wiemy, jakie powinny być wyniki, potrafimy określić wielkości błędów:

$$\delta_j^n = t_j^n - y_j^n = t_j^n - y(\mathbf{x}^n; \mathbf{w}_j)$$

lub ich kwadratów:

$$(\delta_j^n)^2 = [t_j^n - y_j^n]^2 = [t_j^n - y(\mathbf{x}^n; \mathbf{w}_j)]^2.$$

We wzorach powyższych zaznaczono *explicite*, że odpowiedź sieci na j-tym wyjściu, czyli wartość y_j^n jest funkcją wczytanej wartości $\mathbf{x}^n$ oraz aktualnego wektora wag $\mathbf{w}_j$ związanego z j-tym neuronem warstwy wyjściowej.

Zarówno przy adaline jak i przy madaline korzystamy z reguły *delta*, która zakłada liniowe elementy przetwarzające, czyli że y_j^n został wyznaczony ze wzoru:

$$y_j^n = y(\mathbf{x}^n; \mathbf{w}_j) = \mathbf{w}_j^T \mathbf{x}^n.$$

Oznacza to, że wielkości *delta-kwadrat* są obliczane po prostu ze wzoru:

$$(\delta_j^n)^2 = (t_j^n - \mathbf{w}_j^T \mathbf{x}^n)^2, \quad (j = 1, \ldots, M),$$

gdzie j oznacza numer wyjścia.

Niech E (od ang. *Error*) oznacza ogólnie funkcję błędów, lub funkcję celu którą chcemy minimizować. Oznaczając błąd powstały na j-tym wyjściu symbolem E_j i sumując te błędy po wszystkich wyjściach otrzymujemy:

$$E = \sum_{j=1}^{M} E_j = \sum_{j=1}^{M} (\delta_j^n)^2 = \sum_{j=1}^{M} [\delta_j^n(\mathbf{x}^n; \mathbf{w}_j)]^2. \tag{3.4}$$

Oczywiście $E = E(\mathbf{w}_1, \ldots, \mathbf{w}_M) = \sum_{j=1}^{M} E_j(\mathbf{w}_j)$.

Z określenia błędu według wzoru (3.4) wynika, że zagadnienie minimalizacji wielkości E jako funkcji zmiennych $\mathbf{w}_1, \ldots, \mathbf{w}_M$ rozpada się na szereg niezależnych zadań minimalizacji poszczególnych wektorów $\mathbf{w}_j$ $(j = 1, \ldots, M)$ charakterystycznych dla poszczególnych bloków j składających się z pojedynczych elementów *adaline*.

Każdy z wektorów $\mathbf{w}_j$ występuje tylko w jednym błędzie δ_j, stąd zagadnienie sprowadza się do niezależnego obliczenia poprawki $\mathbf{\Delta w}_j$ dla każdego wektora $\mathbf{w}_j$ – tak, jakby to był pojedynczy *adaline*.

3.4. Jak określać funkcję błędu

Widroff i Hoff przyjmowali, że należy minimizować sumę kwadratów błędów — wg zasady minimum sumy kwadratów.

Reguła ta przetrwała do czasów dzisiejszych i funkcja błędu określona według tej zasady jest najbardziej popularnym kryterium również w czasach dzisiejszych.

Jednak metoda ta w ostatnim dwudziestoleciu była poważnie krytykowana.

Jako alternatywy proponowano[3]:

- sumę wartości bezwzględnych błędów (norma L_1)
- bardziej specjalne funkcje, jak np. funkcja Hubera lub Hampela
- mieszane funkcje, np. funkcja Karayiannisa

$$E = \frac{1}{2}\lambda \sum_{j=1}^{M} (t_j - y_j)^2 + (1 - \lambda) \sum_{j=1}^{M} \Phi_1(t_j - y_j).$$

[3] por. Osowski [12], str. 41–44

W końcowej fazie ma być zapewniona minimalizacja wartości absolutnej błędu. Jako funkcję Φ_1 przyjmuje się

$$\Phi_1(x) = \frac{1}{\beta} \ln[cosh(\beta x)]$$

która przy dużych wartościach β dąży do $|x|$.

3.5. Moduły demonstrujące działanie sieci typu „Adaline”

{Tekst w głównej mierze oparty na dokumentacji MatLab Neural Networks version 3, Rozdział 4}

Bardzo proste moduły demonstracyjne MatLaba pokazują, jak w najprostszym przypadku może komplikować się zagadnienie optymalizacji wektora wag.

Moduły demolin1 i demolin2 pokazują trenowanie sieci typu Adaline, tj. z liniową funkcją aktywacji. Rozpatruje się sieć która zawiera tylko jeden neuron. W rozpatrywany przykładzie cała próba ucząca składa się z dwóch wzorców (osobników): p=[1.0 −1.2]; t=[0.5 1.0]; dla każdego osobnika mamy określoną tylko jedną cechę.

Dla rozważanego problemu jest wykreślana powierzchnia błędu (error surface), oraz kontury tej powierzchni.

Najlepszymi parametrami w i b są te, które wyznaczają minimum powierzchni błędu. Moduł demolin1 znajduje te wartości za pomocą funkcji SOLVELIN, a moduł demolin2 metodą gradientów.

Moduł demolin4 ilustruje działanie sieci w przypadku nieliniowego związku między zmienną wejściową x a zmienną docelową d. Przykładowo:
P=[+1.0 +1.5 +3.0 −1.2];
T=[+0.5 +1.1 +3.0 −1.0];

W przypadku nieliniowej zależności błąd nigdy nie osiągnie wartości 0; jednak możemy zbliżyć się w rozsądnych granicach do przybliżonego rozwiązania.

Dla wymienionego problemu ustala się liczbę iteracji (czyli maksymalną liczbę epok) net.trainParam.epochs= 15.

Użytkownik podaje za pomocą funkcji *ginput* wartości początkowe $w^{(0)}, b^{(0)}$. Następnie, posługując się reguła Widrowa i Hoffa, sieć „uczy się”, czyli wyznacza wartości w, b dające coraz to mniejszy błąd dopasowania. Można zauważyć, że po kilku epokach błąd ten zaczyna się stabilizować.

Moduł demolin5 ilustruje działanie sieci w przypadku, gdy układ parametrów nie jest określony w sposób jednoznaczny (ang. *underdetermined problem*), tzn. istnieje nieskończona liczba rozwiązań dla danego problemu.

Przykład: p=[+1.0]; t=[+0.5]; Każda para wartości w, b spełniająca równość:

$$b + wp = 0.5$$

jest rozwiązaniem dającym minimum błędu. Powierzchnia błędu (przedstawiona jako funkcja parametrów w i b) zawiera całą dolinę (rynnę) optymalnych rozwiązań.

Moduł demolin6 ilustruje działanie sieci w przypadku gdy przedstawiane sieci wzorce (próbki) są liniowo zależne, ale wartości docelowe nie odzwierciedlają tej zależności.

Przykład: Próbki są dwuelementowe ($\mathbf{x} = (x_1, x_2)$), i mamy 3 takie próbki; zmienna docelowa jest jedna:

p=[1.0 2.0 3.0; 4.0 5.0 6.0]; t=[0.5 1.0 −1.0];

Tutaj w przedstawionym zestawie uczącym elementy środkowe (druga kolumna) w tablicy p są średnimi arytmetycznymi elementów kolumn 1 i 3; jednak zależność ta nie zachodzi elementu środkowego tablicy t.

Znowu: nie można w tym przypadku zejść z błędem do zera; niemniej jednak sieć typu *adaline* — wykorzystująca przy uczeniu regułę Widrowa i Hoffa — znajduje przybliżone rozwiązanie.

Moduł demolin7 ilustruje działanie sieci w przypadku uczenia się ze zbyt dużym współczynnikiem uczenia (*Too large a learning rate*).

Pokazane to jest na przykładzie: p=[+1.0 −1.2]; t=[+0.5 +1.0];

Dalsze dwa moduły ilustrują adaline sprzężoną z blokiem *tapped delay line*.

Moduł demolin8 (*Adaptive linear layer*) pokazuje, jak sieć typu *adaline* z 1 jednym neuronem ukrytym i blokiem opóźniającym o **1** epokę potrafi nauczyć się nieliniowego związku między sygnałem wejściowym $P(t)$ i sygnałem wyjściowym $Y(t)$. W rezultacie, po pewnym czasie, predykcje sygnału wyjściowego $y(t)$ pokrywają się z faktyczną wartością sygnału, a błąd predykcji jest praktycznie równy zeru.

Moduł nnd10nc — ilustruje działanie tzw. adaptacyjnego filtrowania (*adaptive noise cancellation*). Sieć składająca się z jednego *adaline* połączonego z blokiem opóźniającym o R epok potrafi z nakładających się dwóch sygnałów — właściwego i losowych zakłóceń, czyli tzw. białego szumu — odfiltrować zakłócenia.

Moduł ten pokazuje również, jak szybko sieć uczy się prawidłowej predykcji, jeżeli w algorytmie uczenia zostaną uwzględnione: współczynnik uczenia i momentum.

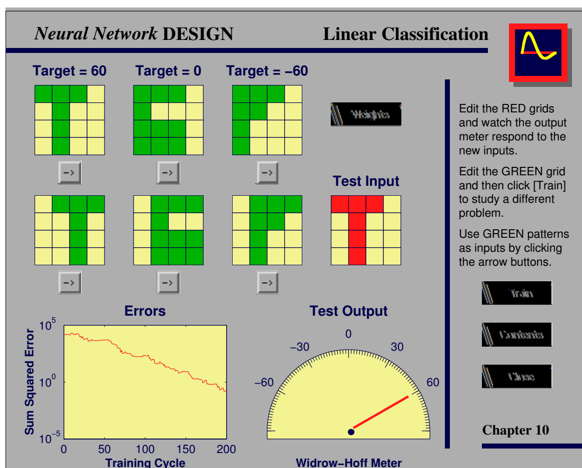
Sieć o podobnej architekturze może służyć do filtrowania (oczyszczenia z obcych sygnałów) np. transmisji głosu pilota przekazywanego w obecności szumu wytwarzanego przez maszynę. Inny przykład, to odfiltrowywania echa (pogłosu) powstającego na łączach linii telefonicznych.

Moduł nnd10lc powtarza demonstrację zaplanowaną przez Widrowa i Hoffa (por. [3], str. 10–37). Demonstracja ta jest oparta na 1 elemencie adaline, który potrafi nauczyć się rozpoznawać 3 litery (np. A, G, F). Próbka ucząca składa się z 3 liter, danych w postaci właściwej i przesuniętej. Kwadrat zawierający każdą literę został podzielony na 16 mniejszych kwadracików, z których niektóre zostały wyróżnione

kolorem zielonym. Tym samym każda próbka litery może być odczytana jako ciąg zero-jedynkowy, w którym „1” odpowiada pokolorowaniu na zielono.

Przy danych wzorcach literek należy najpierw wytrenować sieć, wciskając przycisk train. Po wytrenowaniu można obejrzeć wagi wciskając przycisk weights – ukazuje się wtedy diagram Hintona.

Jeżeli sieć wykształciła już swoje wagi (tzn. nauczyła się rozpoznawać 3 klasy znaków), to można kazać sieci rozpoznać literę podaną w polu „Test input”. Umieszczona na liczniku Widrowa-Hoffa wskazówka pokaże, do której z 3 wyuczonych klas znaków litera umieszczona w polu „Test input” jest najbardziej podobna.



Rysunek 3.2: *Demo nn10lc, Learning classification lc1*

```
  -- Niektóre instrukcje z~DEMHINT  (Netlab)
>> nin=3; nhidden=5; nout=1;
>> net = mlp(nin, nhidden, nout, 'linear');
>> net
net =
       type: 'mlp'
        nin: 3
    nhidden: 5
       nout: 1
       nwts: 26
      actfn: 'linear'
         w1: [3x5 double]
         b1: [-0.1532 -0.0432 -0.6800 -0.0486 -0.6285]
         w2: [5x1 double]
         b2: 0.2423
>> net.w1
ans =
   -1.5304   -0.3875   -0.2240   -0.5037   -0.7243
    0.3973   -0.0186   -0.0054    0.6721    0.9509
   -0.0418   -0.6402   -0.2771   -0.3750    0.3447
>> net.w2
ans =
   -0.2037
    0.3184
    0.3751
    0.0023
   -0.3220
>> [h1, h2] = mlphint(net);
>> h1
h1 =   1
>> h2
h2 = 2
>> disp('The MLP has been created with')
The MLP has been created with
>> disp(['    ' int2str(nin) ' inputs'])
    3 inputs
>> disp(['    ' int2str(nhidden) ' hidden units'])
    5 hidden units
>> disp(['    ' int2str(nout) ' outputs'])
    1 outputs
>> disp(' ')
>> disp('One figure is produced for each layer of weights.')

>> disp('For each layer the fan-in weights are arranged in rows for each unit.')

>> disp('The bias weight is separated from the rest by a red vertical line.')

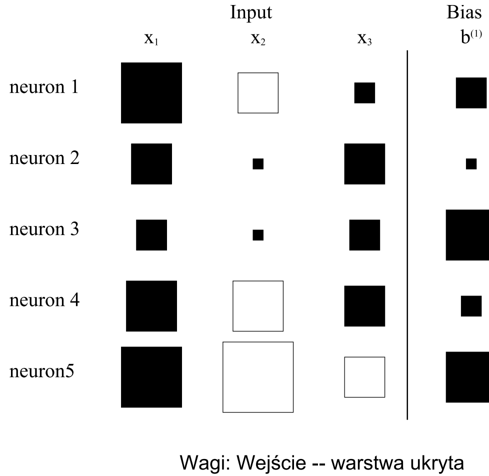
>> disp('The area of each box is proportional to the weight value: positive')

>> disp('values are white, and negative are black.')

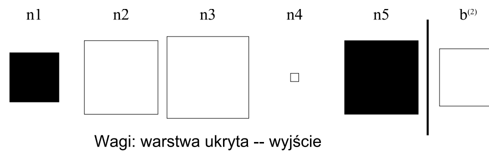

    ===================================
```

3.5.1. Diagramy Hintona – proc. DEMHINT z Netlaba

Pola kwadratów są proporcjonalne do wielkości wag.
Czarne kwadraty — wagi ujemne; białe kwadraty — wagi dodatnie.



Rysunek 3.3: *Wykres Hintona przedstawiający wagi pierwszej warstwy: nin=3, nhidden=5*



Rysunek 3.4: *Wykres Hintona przedstawiający wagi dla drugiej warstwy : nhidden=5 nout=1*

```
function demhint(nin, nhidden, nout)
%DEMHINT Demonstration of Hinton diagram for 2-layer feed-forward network.
%
% Description
%
% DEMHINT plots a~Hinton diagram for a~2-layer feedforward network with
% 5 inputs, 4 hidden units and 3 outputs. The weight vector is chosen
% from a~Gaussian distribution as described under MLP.
%
% DEMHINT(NIN, NHIDDEN, NOUT) allows the user to specify the number of
```

```
% inputs, hidden units and outputs.
%
% See also
% HINTON, HINTMAT, MLP, MLPPAK, MLPUNPAK
%

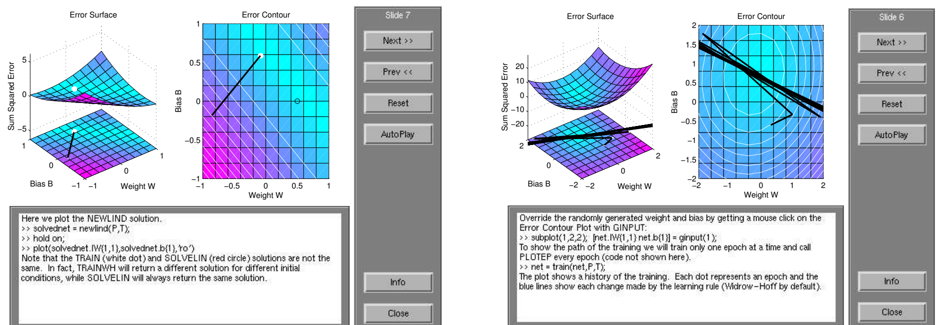
% Copyright (c) Ian T Nabney (1996-9)

if nargin < 1 nin = 5; end
if nargin < 2 nhidden = 7; end
if nargin < 3 nout = 3; end

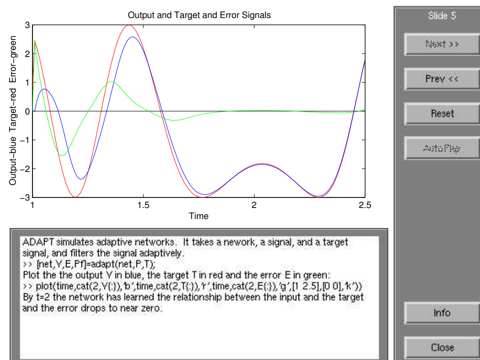
% Fix the seed for reproducible results
randn('state', 42);
clc
disp('This demonstration illustrates the plotting of Hinton diagrams')
disp('for Multi-Layer Perceptron networks.')
disp(' ')
disp('Press any key to continue.')
pause
net = mlp(nin, nhidden, nout, 'linear');

[h1, h2] = mlphint(net);
clc
disp('The MLP has been created with')
disp(['    ' int2str(nin) ' inputs'])
disp(['    ' int2str(nhidden) ' hidden units'])
disp(['    ' int2str(nout) ' outputs'])
disp(' ')
disp('One figure is produced for each layer of weights.')
disp('For each layer the fan-in weights are arranged in rows for each unit.')
disp('The bias weight is separated from the rest by a~red vertical line.')
disp('The area of each box is proportional to the weight value: positive')
disp('values are white, and negative are black.')
disp(' ')
disp('Press any key to exit.');
pause;
delete(h1);
delete(h2);
```

3.5.2. Przykłady grafiki z modułów demonstracyjnych:



Lewa: Demolin5, underdetermined problem. Powierzchnia błędu zawiera dolinę. Czerwone (puste, odizolowane) kółko oznacza rozwiązanie LSE. Plik lin5.eps.
Prawa: Demolin7, Too large a learning rate. Rozwiązanie odbija się od powierzchni błędu, zamiast schodzić do minimum. Proces szukania minimum przy zbyt dużym współczynniku uczenia może być rozbieżny. Plik lin7



Demolin8, Sieć zawierająca na wejściu blok opóźniający sygnał potrafi nauczyć się prognozować nadchodzący sygnał z błędem równym zero. Plik lin8c

Rozdział 4

Zagadnienia dyskryminacji i klasyfikacji

Przez zagadnienia dyskryminacji rozumiemy zagadnienie budowy funkcji dyskryminacyjnych pozwalających na różnicowanie między kilkoma grupami danych.

Zagadnienia klasyfikacji polegają na określaniu przynależności wektorów danych do poszczególnych klas, które są z góry zadane.

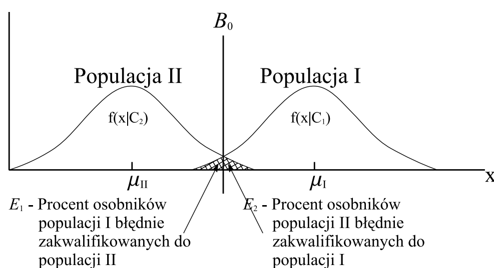
4.1. Dyskryminacja i klasyfikacja dla dwóch grup danych

4.1.1. Przypadek jednej zmiennej

Przypuśćmy, że mamy do czynienia z dwoma populacjami nazywanymi również klasami, oznaczanymi umownie $C1$ i $C2$. Każdy osobnik tej populacji jest charakteryzowany tylko jedną cechą nazywaną X. Rozkład tej cechy w obu populacjach jest inny, i generalnie jest opisany pewną funkcją $f(x|C1)$ w populacji pierwszej i funkcją $f(x|C2)$ w populacji drugiej. W przypadku, gdy cecha X jest cechą numeryczną typu ciągłego, funkcja f nosi nazwę *funkcji gęstości prawdopodobieństwa*. Przykładowe rozkłady funkcji f są pokazane na rysunku 4.1.

Funkcje gęstości prawdopodobieństwa pokazane na tym rysunku bardzo przypominają rozkłady normalne, charakteryzujące się tą samą wariancją σ^2 ale różnymi wartościami oczekiwanymi μ_1 i μ_2.

Przypuśćmy, że naszym celem jest diagnozowanie przynależności do grupy 1, oznaczającej — np. dla ustalenia uwagi — osoby ze stwierdzoną chorobą niedokrwienną serca (ChNS). Natomiast populacja II (Klasa C2) niech oznacza osoby, u których tej choroby nie stwierdzono.



Rysunek 4.1: *Rozkłady cechy X w dwóch populacjach i błędy klasyfikacji, gdy granicą decyzyjną jest wartość B_0. Na osi y wykreślono gęstości p-stw.*

Niech cecha X oznacza np. poziom cholesterolu stwierdzony w surowicy krwi u danej osoby. Naszym celem jest diagnoza ChNS.

Generalnie, rozkłady badanej cechy w obu badanych populacjach mogą częściowo nachodzić na siebie — taki przypadek pokazano na rysunku 4.1.

Naturalną zasadą klasyfikacji jest przydzielenie (zdiagnozowanie) osobnika $\mathbf{x}$ do tej klasy k, dla której wartość $f(\mathbf{x}|C_k)$, $k = 1, 2$, czyli wartość funkcji gęstości wyznaczonej dla tego osobnika, jest największa. Jest to zasada, która w statystyce nosi nazwę zasady *największej wiarygodności*.

Jeśli obie funkcje gęstości nachodzą częściowo na siebie (populacje nie są liniowo separabilne), to z natury rzeczy przy diagnozie możemy popełnić pewne błędy: błąd E_1, że osobnika należącego do populacji 1 (klasy C1) zaklasyfikujemy do populacji 2, i błąd E_2, że osobnika z klasy C2 zaklasyfikujemy do C1.

DOBROĆ REGUŁY KLASYFIKACYJNEJ *mierzymy oczekiwaną frakcją błędnie zaklasyfikowanych osobników. Im ta frakcja jest mniejsza, tym reguła jest lepsza.*

4.1.2. Klasyfikacja na podstawie jednej cechy o rozkładzie normalnym z tą samą wariancją w obu populacjach

Rozpatrzmy bardziej szczegółowo przypadek, gdy rozkłady w obu populacjach są normalne, z wartościami oczekiwanymi μ_1 i μ_2 i **tą samą wariancją** σ^2. Jest to przypadek przedstawiony na rysunku 4.1.

Wyznaczmy najpierw granicę decyzyjną B_0, rozdzielającą obydwa rozkłady. Wielkości błędów E_1 i E_2 są wtedy obliczane jako odpowiednie całki (pola pod krzywymi odpowiednich funkcji gęstości) począwszy od wartości B_0 w lewo lub w prawo.

Punkt $x = B_0$ wyznaczamy jako wartość, dla której gęstości rozkładów $f(x|C1)$ i $f(x|C2)$ są takie same. Mamy więc równość

$$\frac{1}{\sqrt{2\pi}\sigma} exp\{- \frac{(x-\mu_1)^2}{2\sigma^2}\} = \frac{1}{\sqrt{2\pi}\sigma} exp\{- \frac{(x-\mu_2)^2}{2\sigma^2}\} \tag{4.1}$$

Logarytmując obustronnie otrzymujemy

$$- \frac{(x-\mu_1)^2}{2\sigma^2} = - \frac{(x-\mu_2)^2}{2\sigma^2} \tag{4.2}$$

Po uporządkowaniu wyrażeń, wartością x dla której ta równość jest prawdziwa, okazuje się być

$$x = \frac{\mu_1^2 - \mu_2^2}{2(\mu_1 - \mu_2)} = \frac{\mu_1 + \mu_2}{2}. \tag{4.3}$$

Tak więc granicą decyzyjną B_0 jest tutaj średnia arytmetyczna wartości oczekiwanych obserwowanej wartości x w obydwu populacjach.

Nasza decyzja, czy przy danym x zaliczyć osobnika do klasy 1 czy też drugiej opiera się na następującej zasadzie:

$\mu_1 > \mu_2$: Jeśli $x > (\mu_1 + \mu_2)/2$, to zaliczamy osobnika do klasy 1, w przeciwnym przypadku do klasy 2.

$\mu_1 < \mu_2$: Jeśli $x < (\mu_1 + \mu_2)/2$, to zaliczamy osobnika do klasy 1, w przeciwnym przypadku do klasy 2.

W przypadku, gdy $x = (\mu_1 + \mu_2)/2$, nasza decyzja może być oparta na jakimś zrandomizowanym algorytmie, czego tu nie omawiamy.

Podkreślmy jeszcze raz, zasada ta jest słuszna tylko wtedy, gdy obydwie populacje mają tę samą wariancję.

4.1.3. Klasyfikacja w przypadku rozkładów wielowymiarowych

Przenoszą się tu zasady postępowania wprowadzone dla danych jednowymiarowych. Jeżeli potrafimy określić rozkład $f(\mathbf{x}|C_k)$ opisujący rozkład wektora badanych cech $(\mathbf{x})$ w populacji C_k, $k = 1, 2$, to zaliczamy osobnika $\mathbf{x}$ do tej populacji, dla której $f(\mathbf{x}|C_k)$ jest największe.

W przypadku 2 grup osobnika $\mathbf{x}$ zaliczamy do populacji 1, gdy

$$f(\mathbf{x}|C_1) \; > \; f(\mathbf{x}|C_2). \tag{4.4}$$

Pokażemy teraz, że w przypadku gdy $f(\mathbf{x}|C_k)$ przedstawia wielowymiarowy rozkład normalny, zasadę klasyfikacji wyrażoną nierównością 4.4 możemy przedstawić jako warunek liniowy względem rozważanych zmiennych objaśniających.

Przypomnijmy najpierw, że funkcja gęstości d-wymiarowego rozkładu normalnego ma postać:

$$f(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{\frac{1}{2}d}|\boldsymbol{\Sigma}|^{\frac{1}{2}}} \times \exp\{-\frac{1}{2}[\,(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\,]\},$$

gdzie

$\mathbf{x} = (x_1, x_2, \ldots, x_d)^T$ jest wektorem danych,

$\boldsymbol{\mu} = (\mu_1, \mu_2, \ldots, \mu_d)^T$ jest wektorem wartości oczekiwanych,

$\boldsymbol{\Sigma}_{d\times d}$ jest macierzą kowariancji między zmiennymi $X_1, X_2, ..., X_d$ (macierz symetryczna, na przekątnej znajdują się wariancje tych zmiennych, a poza przekątną kowariancje).

Załóżmy, że rozkład ten dla populacji 1 i populacji 2 jest charakteryzowany wartościami oczekiwanymi $\boldsymbol{\mu}_1$ i $\boldsymbol{\mu}_2$ odpowiednio. Obydwa rozkłady mają tę samą macierz kowariancji.

Naszą zasadą jest, aby osobnika $\mathbf{x}$ zaliczyć do populacji 1 wtedy, gdy

$$f(\mathbf{x};\boldsymbol{\mu}_1,\boldsymbol{\Sigma}) \;>\; f(\mathbf{x};\boldsymbol{\mu}_2,\boldsymbol{\Sigma}). \tag{4.5}$$

Podstawiając do tego wzoru odpowiednie wyrażenia określające rozkład normalny w obu populacjach, oraz logarytmując te wyrażenia otrzymujemy:

$$-\frac{1}{2}[\,(\mathbf{x}-\boldsymbol{\mu}_1)^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)\,] \;>\; -\frac{1}{2}[\,(\mathbf{x}-\boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)\,].$$

Przenosząc wszystkie składniki na lewą stronę nierówności, wykonując mnożenia i grupując względem wektora $\mathbf{x}$ otrzymujemy

$$(\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}\mathbf{x} + \{\,-\frac{1}{2}\boldsymbol{\mu}_1^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_2\} > 0. \tag{4.6}$$

Warunek ten możemy zapisać prościej

$$\mathbf{w}^T\mathbf{x} + w_0 > 0, \tag{4.7}$$

gdzie

$$\mathbf{w}^T = (\boldsymbol{\mu}_1-\boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}, \quad w_0 = -\frac{1}{2}\boldsymbol{\mu}_1^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_2. \tag{4.8}$$

Jak widać, w przypadku dwóch rozkładów normalnych granicę decyzyjną stanowi hiperpłaszczyzna $\mathbf{w}^T\mathbf{x} + w_0 = 0$ wyznaczona na podstawie parametrów charakteryzujących obywa rozkłady.

Współczynniki $\mathbf{w}$ i stałą w_0 można obliczyć ze wzoru (4.8) lub też otrzymać za pomocą sieci neuronowej o architekturze perceptronu prostego z jednym neuronem i liniową funkcją aktywacji.

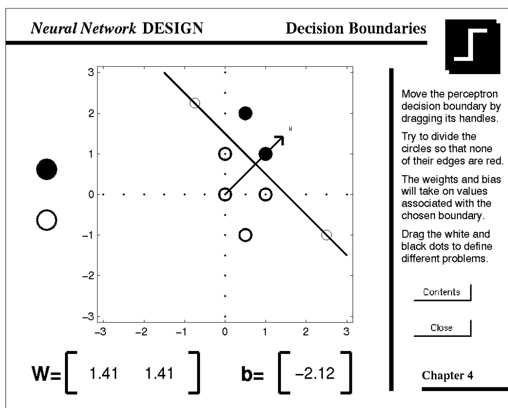
Gdybyśmy mieli do czynienia tylko z dwoma zmiennymi, czyli $\mathbf{x} = (x_1, x2)$, wtedy granicą decyzyjną (decision boundary) rozdzielającą obydwie populacje byłaby prosta:

$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

leżąca w płaszczyźnie (x_1, x_2). Wektor $\mathbf{w} = [w_1, w_2]$ jest wektorem ortogonalnym do tej prostej, a stała w_0 wyznacza wielkość przesunięcia tej prostej względem początku układu.

Ponadto można powiedzieć, że wektor $\mathbf{w}$ wskazuje kierunek populacji C1.

Fakty te można było zaobserwować w module demonstracyjnym nnd4db – decision boundaries. Plansza z tego modułu jest pokazana na rysunku 4.2.



Rysunek 4.2: *Prosta dyskryminacyjna dla klasyfikacji 2 grup oznaczonych czarnymi (I) i białymi (II) kółkami. Równanie prostej rozgraniczającej:* $1.41x_1 + 1.41x_2 - 2.12 = 0$. *Wektor* $w = [1.41, 1.41]$ *jest wektorem ortogonalnym do prostej rozgraniczającej i jest skierowany na tę stronę półpłaszczyzny, po której znajdują się punkty populacji wyróżnionej, u nas: czarne punkty*

4.1.4. Prawo Bayesa i obliczanie prawdopodobieństw á posteriori

Poprzednio chodziło nam ogólnie o zaklasyfikowanie osobnika $\mathbf{x}$ do jednej z dwóch grup danych. Obecnie chodzi nam o coś więcej: mianowicie o określenie *prawdopodobieństwa*, że osobnik scharakteryzowany wektorem $\mathbf{x}$ należy do określonej (k-tej) grupy danych. Prawdopodobieństwo to będziemy nazywać *prawdopodobieństwem á posteriori* i oznaczać $p(C_k \mid \mathbf{x})$.

Prawdopodobieństwo to jest wyznaczane według prawa Bayesa[1].

Prawo to opiera się na dwóch następujących zasadach obowiązujących przy określaniu tzw. prawdopodobieństwa łącznego pary zmiennych ($\mathbf{x}$, C), gdzie C oznacza zmienną określającą podział na kategorie (u nas C przyjmuje wartości $C = \{C_1, C_2\}$), a $\mathbf{x}$ oznacza oznacza przestrzeń obserwowanych zmiennych na podstawie których chcemy dokonać klasyfikacji (zmienne te są czasem nazywane zmiennymi objaśniającymi).

Zakładamy, że zmienna C, tj. klasyfikacja na klasy $\{C_1, C_2\}$ pokrywa całą przestrzeń $\mathbf{x}$.

Rysunek 4.3 ilustruje obliczanie prawdopodobieństw całkowitych i warunkowych.

Para zmiennych (x, C)

C_1 n_1

C_2 n_2

x N

$f(x|C_1)$ – rozkład w górnej warstwie
$f(x|C_2)$ – rozkład w dolnej warstwie
$f(x)$ – rozkład całkowity, bez podziału na warstwy klasowe

Rysunek 4.3: *Rozkład łączny pary zmiennych* $(\mathbf{x}, \mathrm{C})$ *i wynikające z niego rozkłady warunkowe*

W rachunku prawdopodobieństwa są znane dwa podstawowe prawa dotyczące zależności między prawdopodobieństwami całkowitymi, łącznymi i warunkowymi:

Prawo iloczynu (Product rule): $P(C_k, \mathbf{x}) = P(C_k \,|\, \mathbf{x}) \cdot p(\mathbf{x})$

Prawo sumowania (Sum rule): $\sum_k P(C_k, \mathbf{x}) = p(\mathbf{x})$.

Na podstawie tych praw zostało sformułowane tzw. prawo Bayesa:

$$P(C_k \,|\, \mathbf{x}) = \frac{p(\mathbf{x} \,|\, C_k) P(C_k)}{p(\mathbf{x})} \tag{4.9}$$

przy czym $p(\mathbf{x})$ może być obliczane jako

$$p(\mathbf{x}) = \sum_k P(\mathbf{x} \,|\, C_k) P(C_k).$$

[1] Rev. Thomas Bayes, 1702–1761

P-stwa występujące we wzorze Bayesa noszą następujące nazwy:

$P(C_k \,|\, \mathbf{x})$ – prawdopodobieństwo *á posteriori*; mówi ono, jak wysokie jest p-stwo, że osobnik scharakteryzowany wektorem $\mathbf{x}$ został wylosowany z populacji k-tej (the p-blity of an event conditional on the observations);

$p(\mathbf{x} \,|\, C_k)$ – funkcja gęstości p-stwa w populacji C_k, czyli rozkład x uwarunkowany kategorią C_k (conditionned on class $C1$);

$P(C_k)$ – prawdopodobieństwo *á priori*, że dany osobnik pochodzi z k-tej populacji; p-stwo to jest wyznaczane na podstawie innych informacji, bez uwzględniania wektora obserwacji $\mathbf{x}$ dla tego osobnika (probability specified before seeing the data, and so based on prior experience or belief);

$p(\mathbf{x})$ – p-stwo całkowite zmiennej $\mathbf{x}$, bezwarunkowe, tzn. bez uwzględniania podziału na klasy; jest to p-stwo brzegowe dla rozkładu łącznego $(\mathbf{x}, C)$.

Bishop ([1], str. 82–83, również str. 232–233) pokazał, że jeżeli rozkłady warunkowe $p(\mathbf{x} \,|\, C_k)$ należą do tzw. rodziny wykładniczej, to prawidłowo wytrenowana sieć neuronowa podaje na wyjściu właśnie prawdopodobieństwa á posteriori $P(C_k \,|\, \mathbf{x}^n)$:

$$\mathbf{y}^n = \mathbf{y}(\mathbf{w}; \mathbf{x}^n) = (y_1^n, \ldots, y_c^n),$$

gdzie

$$y_k^n \approx P(C_k \,|\, \mathbf{x}^n), \;\; k = 1, \ldots, c.$$

4.1.5. P-stwa á posteriori dla 2 klas i funkcja logistyczna

Dla przypadku gdy liczba klas $c = 2$, a nas interesuje klasa C_1 (np. ChNS), to — używając reguły Bayesa — p-stwo á posteriori należenia osobnika $\mathbf{x}$ do klasy C_1 możemy zapisać następująco:

$$P(C_1 \,|\, \mathbf{x}) = \frac{p(\mathbf{x} \,|\, C_1)P(C_1)}{p(\mathbf{x} \,|\, C_1)P(C_1) + p(\mathbf{x} \,|\, C_2)P(C_2)} = \frac{1}{1 + \exp(-a)}, \tag{4.10}$$

gdzie

$$a = \ln \frac{p(\mathbf{x} \,|\, C_1)P(C_1)}{p(\mathbf{x} \,|\, C_2)P(C_2)}. \tag{4.11}$$

Łatwo pokazać (por. [1], str. 233–234), że jeżeli rozkład $p(\mathbf{x} \,|\, C_1)$ da się przedstawić jako rozkład należący do rodziny wykładniczej, czyli w postaci

$$p(\mathbf{x} \,|\, C_1) = \exp \; \{A(\boldsymbol{\theta}_k) + B(\boldsymbol{x}, \boldsymbol{\phi}) + \boldsymbol{\theta}_k^T \mathbf{x}\}$$

to obliczoną wartość a można przekształcić do następującej

$$a = \mathbf{w}^T \mathbf{x} + w_0,$$

gdzie

$$\mathbf{w} = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_2, \quad w_0 = A(\boldsymbol{\theta}_1) - A(\boldsymbol{\theta}_2) + \ln \frac{P(C_1)}{P(C_2)}.$$

W przypadku rozkładów normalnych parametry $\boldsymbol{\theta}_1$ i $\boldsymbol{\theta}_2$ odpowiadają wartościom oczekiwanym $\boldsymbol{\mu}_1 \boldsymbol{\Sigma}^{-1}$ i $\boldsymbol{\mu}_2 \boldsymbol{\Sigma}^{-1}$ odpowiednio.

Tak więc:
jeśli obliczone dane mają rozkłady $p(\mathbf{x} \,|\, C_1)$ i $p(\mathbf{x} \,|\, C_2)$ należące do rodziny rozkładów wykładniczych,
to obliczenia za pomocą sieci neuronowych z 1 neuronem i logistyczną funkcji aktywacji są równoważne obliczeniom Bayesowskiego p-stwa á posteriori.

Obliczone p-stwo á posteriori $p(C_1 \,|\, \mathbf{x})$ zależy od kombinacji liniowej wartości zmiennych zapamiętanych w wektorze $\mathbf{x}$ ($a = \mathbf{w}^T \mathbf{x}$) za pośrednictwem logistycznej funkcji aktywacji. Statystycy mówią, że jest to uogólniony model liniowy z funkcją pośredniczącą $f(a)$ (link function). W naszym przypadku funkcją pośredniczącą jest funkcja logistyczna.

Z powyższych rozważań wynika, iż w wielu przypadkach należy spodziewać się, że na wyjściu sieci będą pojawiać się wartości y^n które będą — za pośrednictwem funkcji logistycznej — zależały od zmiennej a będącej kombinacją liniową zmiennych wejściowych $x_1, \ldots, x_d$. Mówimy w takim przypadku, że mamy do czynienia z *uogólnionym modelem liniowym* (ang. generalized linear model, GLM).

4.2. Funkcja błędu E i funkcje aktywacji

4.2.1. Funkcja błędu wyprowadzona z zasady największej wiarygodności

Oznaczmy wiarygodność

$$L = \prod_n p(\mathbf{t}^n \mid \mathbf{x}^n)$$

Symbol $p(\cdot)$ oznacza tu prawdopodobieństwo dla zmiennych dyskretnych lub funkcję gęstości p-stwa dla zmiennych ciągłych.

Generalnie $p(\cdot)$ zależy od jakichś parametrów. Parametry te należy tak dobrać, aby zmaksymizować L.

Zamiast maksymizować L możemy minimizować logarytm wiarygodności L wzięty ze znakiem przeciwnym:

$$E = \,-\, \ln L = -\sum_n \ln p(\mathbf{t}^n | \mathbf{x}^n). \tag{4.12}$$

Bardziej konkretne określenie funkcji E zależy od postaci funkcji $p(\mathbf{t}^n \,|\, \mathbf{x}^n)$ oznaczającej prawdopodobieństwo warunkowe wystąpienia zdarzenia losowego $\mathbf{t}$ pod warunkiem zdarzenia $\mathbf{x}$.

4.2.2. Przypadek klasyfikacji do 2 grup

W tym przypadku zmienna $\mathbf{t}$ jest jednowymiarowa i może przyjmować tylko wartości 1 lub 0. Wobec tego $p(\mathbf{t} \,|\, \mathbf{x})$ możemy zapisać w postaci

$$p(\mathbf{t} \,|\, \mathbf{x}) = y^t (1-y)^{1-t},$$

gdzie y oznacza ogólnie prawdopodobieństwo wystąpienia „1".

Powyższy zapis obejmuje zarówno przypadek wystąpienia „1" jak i „0".

Dla n-tego osobnika możemy powyższe zapisać jako

$$p(\mathbf{t}^n \,|\, \mathbf{x}^n) = (y^n)^{t^n} (1-y^n)^{1-t^n}.$$

Wiarygodność zaobserwowania ciągu $\{\mathbf{x}^n, t^n\},\ n = 1, .., N$ wynosi wtedy

$$E = -\sum_{n=1}^{N} \{t^n \ln y^n + (1-t^n)(1-y^n)\} \tag{4.13}$$

Kryterium to nosi nazwę *entropii krzyżowej* (cross-entropy).

Generalnie wartość y^n jest funkcją zmiennych objaśniających $\mathbf{x}^n$ i dodatkowych parametrów opisujących przyjęty model. W przypadku obliczeń przez sztuczne sieci neuronowe parametrami modelu są wagi $\mathbf{w}$:

$$y^n = y(\mathbf{x}^n, \mathbf{w}).$$

W przypadku sieci jednowarstwowej z jednym neuronem i logistycznej funkcji aktywacji wartość y^n jest funkcją zmiennej a^n będącej liniową kombinacją zmiennych wejściowych $\mathbf{x}^n$.

$$\frac{\partial E}{\partial y^n} = \frac{y^n - t^n}{y^n(1-y^n)}$$

The absolute minimum is at $y^n = t^n$ for all n.

$y = g(a)$, gdzie g jest f. logistyczną, to $g'(a) = g(a)(1-g(a))$.

$$\delta^n = \frac{\partial E}{\partial a^n} = \frac{\partial E}{\partial y^n} \cdot \frac{\partial y^n}{\partial a^n} = y^n - t^n$$

Tak więc gradient tutaj określonej funkcji błędu wyznaczony dla wartości $\mathbf{w} = \mathbf{w}^{(k)}$ wynosi

$$\nabla E|_{\mathbf{w}^{(k)}} = \sum_n [y(\mathbf{x}^n; \mathbf{w}^{(k)}) - t^n]\mathbf{x}^n.$$

W pakiecie NETLAB do minimalizacji funkcji błędu określonej wzorem (4.13) używa się metody IRLS (*Iterative Reweighted Least Squares*).

4.2.3. Przypadek klasyfikacji do c grup

W tym przypadku $\mathbf{t} = [t_1, \ldots, t_c]$ jest wektorem o c składowych, spełniającym restrykcje: $t_k \in \{0,1\}$; $\sum_k t_k = 1$. Funkcja wiarygodności otrzymania wektora $\mathbf{t}$ pod warunkiem zaobserwowania $\mathbf{x}$ wyraża się wzorem

$$L = p(\mathbf{t} \mid \mathbf{x}) = \prod_k y_k^{t_k},$$

gdzie $\sum_k y_k = 1$. Wzór ten wynika z określenia rozkładu multinomialnego. Wartości y_k wyrażają prawdopodobieństwa á posteriori, że osobnik scharakteryzowany wektorem $\mathbf{x}^n$ będzie należał do k-tej klasy.

Ujmując to jeszcze inaczej, możemy powiedzieć że symbol y_k oznacza tu ogólnie p-stwo pojawienia się „1" w k-tej klasie ($k = 1, \ldots, c$). Prawdopodobieństwo to może być różne dla różnych osobników i zależeć w szczególności od jego wartości $\mathbf{x}$, oraz od dodatkowych parametrów, którymi w przypadku obliczeń za pomocą sieci neuronowych są wagi $\mathbf{w}$:

$$\mathbf{y}^n = \mathbf{y}(\mathbf{x}^n; \mathbf{w})$$

Jeżeli rozpatrujemy próbkę N osobników charakteryzującymi się wartościami $\mathbf{x}^n$ oraz $\mathbf{t}^n$, $n = 1, \ldots, N$, to wiarygodność otrzymania takiej próbki możemy zapisać:

$$L = \prod_n p(\mathbf{t}^n \mid \mathbf{x}^n) = \prod_n \prod_k (y_k^n)^{t_k^n}. \tag{4.14}$$

Wagi $\mathbf{w}$ i bias b_0 mają być wybrane tak, aby zmaksymizować wiarygodność.

Logarytmując i biorąc wynik logarytmowania ze znakiem ujemnym, otrzymujemy

$$E = -lnL = -\sum_n \sum_k t_k^n \cdot \ln(y_k^n). \tag{4.15}$$

4.2.4. Minimalizacja błędu

Określone wyżej funkcje błędu dla klasyfikacji do 2 lub $c > 2$ klas określone wzorami (4.13) i (4.15) zależą od wag $\mathbf{w}$ które należy wyznaczyć iteracyjnie w procesie uczenia (treningu) sztucznej sieci neuronowej.

Funkcja błędu E jest — poza nielicznymi przypadkami – funkcją nieliniową wektora wag — wobec czego nie można wag wyznaczyć bezpośrednio. Funkcja ta jest czasami nazywana *funkcją celu.*

Jeżeli E jest różniczkowalna, to możemy stosować jakieś metody gradientowe. Zasada jest taka, że dysponując jakimś wektorem $\mathbf{w}(k)$ otrzymanym w k-tym kroku, „poprawiamy" ten wektor poruszając się o małą odległość w kierunku, w którym funkcja E maleje najbardziej, czyli w kierunku ujemnego gradientu, tj. w kierunku $-\nabla_w E$.

W ten sposób w kolejnych krokach k mamy nadzieję zbliżać się do minimum funkcji E. Ciąg kolejnych przybliżeń konstruujemy wg zasady:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \left.\frac{\partial E}{\partial \mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}^{(k)}}.$$

Współczynnik η nosi nazwę współczynnika uczenia (*learning rate*).

Jednak — metoda ta może zaprowadzić nas do lokalnego minimum, i w praktyce jest znanych i stosowanych wiele innych algorytmów optymalizacyjnych. Przykładowo, w pakiecie Netlab można używać takich metod, jak metoda sprzężonych gradientów (conjugate gradients), metoda skalowanych sprzężonych gradientów (scaled conjugate gradients), metoda quasi-Newton.

4.2.5. Funkcje aktywacji używane w zagadnieniach klasyfikacyjnych

Generalnie używa się tu 3 typów funkcji: liniowej, logistycznej i softmax.

Funkcja logistyczna ma postać:

$$y = \frac{1}{1 + e^{-a}} ,$$

gdzie $a = \mathbf{w}^T \mathbf{x}$ oznacza wynik sumowania bodźców przez neuron.

Funkcja softmax jest określona w kontekście kilku wyjść, czyli warstwy wyjściowej składającej się z kilku neuronów sumujących bodźce wytwarzane przez poprzednią warstwę.

Niech $a_k = \mathbf{w}_k^T \mathbf{x}$ oznacza wynik sumowania wykonany przez k-ty neuron, $k = 1, \ldots, c$. Wtedy aktywacja k-tego neuronu za pomocą funkcji softmax jest określona następująco:

$$y_k = \frac{e^{a_k}}{\sum_k e^{a_k}}. \tag{4.16}$$

Jak widzimy, funkcja *softmax* zapewnia nam, że $\sum_k y_k \equiv 1$, co oznacza, że wyniki y_k mogą być interpretowane jako prawdopodobieństwa á posteriori.

4.3. Procedury klasyfikacyjne w pakiecie Netlab

Pakiet Netlab — zaimplementowany jako zestaw funkcji w MatLabie w postaci tzw. m-files — posiada implementację kilku modeli klasyfikacyjnych: GLM, MLP, KNN i GMM. Modele te (za wyjątkiem KNN) są przetwarzane jako pewne specyficzne struktury danych. Ułatwia to dostęp do odpowiednich informacji i zapobiega podstawianiu przez użytkownika modelu niewłaściwych funkcji.

Każdy z wymienionych modeli (za wyjątkiem KNN) posiada funkcję typu *constructor*, który tworzy i inicjuje odpowiednią dla tego modelu strukturę danych

(obiekt) o nazwie net z odpowiednim trzy-literowym przedrostkiem (np. glmnet, mlpnet, gmmnet) który używany jest we wielu towarzyszących modelowi funkcjach.

Praca modelu, a przede wszystkim otrzymywane jako rezultat tej pracy wyniki, są uwarunkowane tzw. *opcjami*. Opcje te są zapamiętywane w tablicy options(1:18). W systemie Netlab jest zainicjowana tablica foptions(1:18) o wartościach domyślnych. Użytkownik powinien skopiować tę tablicę do swojej tablicy options, a następnie — już na gruncie swoich 'options' — nadać poszczególnym elementom odpowiednie wartości. Najczęściej wykorzystywane są następujące opcje dotyczące procesu uczenia (train):

options(1) – wartość 1 oznacza drukowanie błędu po wykonaniu każdej iteracji;
options(2) – wymagana dokładność wag;
options(3) – wymagana dokładność funkcji błędu E;
options(14) – maksymalna liczba iteracji.

4.3.1. Metoda GLM — czyli uogólnionego modelu liniowego

Model ten opisuje sieć jednowarstwową; warstwa ta jest jednocześnie warstwą wyjściową (warstwa wejściowa nie liczy się). Tworzenie struktury:

net = glm(n_in, n_out, actfn) lub
net = glm(input_dim, n_out, actfn, prior)

Struktura 'net' przy modelu GLM:

type	'glm'
n_in	liczba wejść (liczba cech danych)
n_out	liczba wyjść sieci
nwts	liczba wszystkich współczynników wagowych i biasów
actfn	f. aktywacji: string 'linear', 'sigmoid', 'softmax'
w1, b1	tablica wag i biasów występujących w pierwszej warstwie (jedynej)

W przypadku używania złożonego modelu Bayesowskiego struktura glmnet przewiduje dalsze pola określające parametry Bayesowskie: pola prior i beta.

Dalsze czynności to trenowanie sieci i sprawdzenie jej skuteczności (tj. możliwości klasyfikacji tych samych lub innych danych testowych):

[net, options]= glmtrain(net, options, data, targets)
[y, a] = glmfwd(net, testdata);

Przykładowy skrypt wykonujący klasyfikację za pomocą metody GLM:

```
net=glm(3,1,'logistic');
data=w65(:,[2 3 4]);  % bierzemy tylko kolumny 2, 3, 4
target=w65(:,5);      % bierzemy kolumne 5 tablicy z~danymi
options=zeros(1,18); options(1)=1; options(14)=30;
net=glmtrain(net,options,data, target);
tdata=gr7(:,[3 5 6]);  % dane testowe
y=glmfwd(net, tdata);  % wyniki sieci dla danych testowych
```

4.3.2. Metoda MLP — dla dwuwarstwowego perceptronu

Implementacja w Netlabie uwzględnia tylko perceptrony dwuwarstwowe, tj. zawierające jedną warstwę ukrytą i jedną warstwę wyjściową. Tworzenie struktury:

net = mlp(n_in, n_hidden, n_out, actfn)

lub też — w przypadku używania złożonego modelu Baysowskiego:

net = mlp(input_dim, n_hidden, n_out, actfn, prior,beta)

Struktura 'net' przy modelu MLP:

type	'mlp'
n_in	liczba wejść (liczba cech danych)
n_hidden	l. neuronów w warstwie ukrytej
n_out	liczba wyjść sieci
nwts	liczba wszystkich współczynników wagowych i biasów
actfn	f. aktywacji: string 'linear', 'sigmoid', 'softmax'
w1, b1,w2,b2	tablica wag i biasów występujących w obu warstwach

Struktura może zawierać dalsze pola określające parametry złożonego modelu Baysowskiego.

Dalsze czynności to trenowanie sieci i sprawdzenie jej skuteczności (tj. możliwości klasyfikacji):

[net, options, varargout]= netopt(net, options, x, t, alg);

[e, edata, eprior] = mlpfwd(net, testdata);

Przykładowy skrypt wykonujący klasyfikację za pomocą metody MLP:

```
nh=5;     % l. neuronow w~warstwie ukrytej
net=mlp(3,nh,1,'logistic');
data=w65(:,[2 3 4]);  target=w65(:,5);
options=zeros(1,18); options(1)=1; options(14)=30;
[net, options]=netopt(net, options, data, target, 'conjgrad');
      % innym optymizerem moze byc np.: 'scg' lub 'quasinew'
tdata=gr7(:,[3 5 6]);
options5=options;    % to jest na ogol niepotrzebne
y_mlp5=mlpfwd(net,tdata);
% tab=[gr7(:,1) y_mpl1 y_mlp3 y_mlp5];
```

4.3.3. Metoda KNN — najbliższych K sąsiadów

4.3.4. Metoda GMM — bazująca na mieszaninie rozkładów

Rozdział 5

Algorytmy minimalizacji błędu

5.1. Znajdywanie minimum funkcji błędu

Funkcja błędu $E = E(\mathbf{w})$ jest w ogólnym przypadku nieliniową funkcją wektora wag $\mathbf{w} = [w_1, w_2, \ldots, w_S]$, gdzie S jest sumaryczną liczbą wag łącznie z wartościami progowymi (biasami). Funkcja E przedstawia najczęściej sumę kwadratów różnic między wartościami docelowymi (t^n) a wartościami $y^n(\mathbf{x}^n; \mathbf{w})$ wyprodukowanymi przez sieć; czasami — szczególnie w zagadnieniach klasyfikacyjnych — jest to ujemny logarytm z prawdopodobieństw aposteriori lub z wiarygodności $P(\mathbf{t}^n|bfx^n)$ otrzymania wartości docelowych t^n pod warunkiem zaobserwowania wektora zmiennych objaśniających $\mathbf{x}^n$.

5.2. Klasy algorytmów minimalizacyjnych

Generalnie możemy tu wyróżnić trzy klasy:

1. Algorytmy klasyczne. Są to algorytmy związane z zagadnieniami minimalizacyjnymi w ogóle. Nie wykorzystują specyfiki sieci neuronowych i są stosowane w wielu innych zagadnieniach optymalizacyjnych. Najczęściej są to algorytmy gradientowe, chociaż są znane również algorytmy bezgradientowe.
2. Algorytmy związane specyficznie z sieciami neuronowymi. Zalicza się tu przede wszystkim algorytm tzw. *propagacji wstecznej* czyli *backpropagation*. Algorytm backpropagation bierze pod uwagę rozkład neuronów (a tym samym przypisanych im wag) w poszczególnych warstwach.

3. Algorytmy z elementami stochastycznymi. Należą tu w pierwszym rzędzie: algorytm symulowanego wyżarzania, algorytmy genetyczne i algorytmy ewolucyjne.

W dalszym ciągu omówimy jedynie kilka algorytmów klasycznych wykorzystujących metody gradientowe.

5.3. Algorytmy klasyczne służące minimalizacji funkcji $E(w)$

Generalnie rzecz biorąc, są to algorytmy iteracyjne.

5.3.1. Zasada algorytmów iteracyjnych

Z jakiegoś początkowego przybliżenia wektora $\mathbf{w}^{(0)}$ konstruujemy w kolejnych krokach (iteracjach) ciąg kolejnych przybliżeń wektora $\mathbf{w}$ który ma dać minimum funkcji błędu $E(\mathbf{w})$. Przypuśćmy, że po k krokach otrzymaliśmy wartość $\mathbf{w}^{(k)}$. W następnym kroku, $(k+1)$-szym, konstruujemy następne przybliżenie, $(k+1)$-sze, według zasady:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \Delta\mathbf{w}^{(k)}. \tag{5.1}$$

Cała sztuka algorytmu leży w tym, aby w skuteczny sposób znaleźć poprawkę $\Delta\mathbf{w}^{(k)}$. Poprawka ta polega na ogół na znalezieniu pewnego wektora $\mathbf{p}^{(k)}$ i dodaniu części tego wektora do posiadanego aktualnie wektora $\mathbf{w}^{(k)}$ (nie możemy dodać za dużo, bo moglibyśmy „przeskoczyć” właściwe minimum). Tak więc podstawiamy

-
$$\Delta\mathbf{w}^{(k)} = \eta\mathbf{p}^{(k)}. \tag{5.2}$$

  Dalej poznamy różne metody znajdywania kierunku $\mathbf{p}$. W najprostszym przypadku może to być ujemny gradient funkcji E w punkcie $\mathbf{w}^{(k)}$, co oznacza że podstawiamy $\mathbf{p}^{(k)} = -\nabla E|_{w^{(k)}}$. Wtedy poprawka $\Delta\mathbf{w}^{(k)}$ przyjmie postać:

$$\Delta\mathbf{w}^{(k)} = -\eta\nabla E\Big|_{\mathbf{w}^{(k)}}. \tag{5.3}$$

  Współczynnik η nosi nazwę współczynnika uczenia. Powinien on przyjmować wartości z przedziału [0, 1]. Na ogół wprowadza się go jako funkcję malejącą numeru iteracji.

  Jeżeli wartości η są zbyt małe, to postęp w zbliżaniu się do minimum jest zbyt wolny; jeżeli η jest zbyt duże, to wartości $\mathbf{w}$ mogą zmieniać się zbyt chaotycznie.

- Aby przeciwdziałać zbyt chaotycznym zmianom, wprowadza się do wzoru na $\Delta\mathbf{w}^{(k)}$ dodatkowy człon, tzw. *momentum* lub *pęd*:

$$\Delta\mathbf{w}^{(k)} = \underbrace{-\eta\,\nabla E\Big|_{\mathbf{w}^{(k)}}}_{gradient} + \rho\underbrace{\left(\mathbf{w}^{(k)} - \mathbf{w}^{(k-1)}\right)}_{momentum} . \tag{5.4}$$

  Dodatkowy człon *momentum* zapobiega raptownym zmianom kolejno wyznaczonych wartości $\mathbf{w}^{(k)}$; można powiedzieć, że do pewnego stopnia zachowuje kierunek (tendencję) zmian.

  Określenie odpowiednich wartości η i ρ jest sprawą wyczucia i doświadczenia. Technika *bold driver technique* ([1], str. 269), pozwala wyznaczyć jednocześnie współdziałające wartości η i ρ.

5.3.2. Aproksymacja funkcji błędu za pomocą rozwinięcia w szereg Taylora — i co z tego wynika

Rozwinięcie funkcji $E(\mathbf{w}+\mathbf{p})$ dookoła punktu $\mathbf{w}$:

$$E(\mathbf{w}+\mathbf{p}) = E(\mathbf{w}) + \mathbf{g}(\mathbf{w})^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{H}(\mathbf{w})\mathbf{p} + O(h^3), \tag{5.5}$$

gdzie:

$$\mathbf{g}(\mathbf{w}) = \nabla E\Big|_{\mathbf{w}} = \left[\frac{\partial E}{\partial w_1}\Big|_{\mathbf{w}}, \dots, \frac{\partial E}{\partial w_S}\Big|_{\mathbf{w}}\right]^T, \tag{5.6}$$

$$\mathbf{H} = \left(\frac{\partial^2 E}{\partial w_i \partial w_j}\right)\Big|_{\mathbf{w}}, \quad i,j = 1,\dots,S. \tag{5.7}$$

Wyrażenie $\mathbf{g}(\mathbf{w})$ nazywa się gradientem funkcji (powierzchni) E w punkcie $\mathbf{w}$ i określa płaszczyznę styczną do tej powierzchni.

Macierz $\mathbf{H}$ nazywa się hesjanem i zawiera w sobie informacje o krzywiźnie powierzchni.

Różne algorytmy służące minimizacji w dużym stopniu korzystają z przedstawionego wyżej rozwinięcia (5.5); przy czym niektóre odmiany tych algorytmów biorą tylko człony liniowe, a inne zarówno człony liniowe jak i kwadratowe.

5.3.3. Algorytm największego spadku

W rozwinięciu (5.5) ograniczamy się tylko do członu liniowego rozwinięcia:

$$E(\mathbf{w}^{(k)} + \mathbf{p}^{(k)}) = E(\mathbf{w}^{(k)}) + \mathbf{g}(\mathbf{w}^{(k)})^T\mathbf{p}^{(k)} + O(h^2). \tag{5.8}$$

Chcemy, żeby

$$E(\mathbf{w}^{(k+1)}) < E(\mathbf{w}^{(k)}).$$

Będzie tak, gdy

$$\mathbf{g}(\mathbf{w}^{(k)})^T \mathbf{p}^{(k)} < 0\,.$$

Wynika stąd, że będzie tak (tj. nastąpi obniżenie błędu), gdy przyjmiemy:

$$\mathbf{p}^{(k)} = -\mathbf{g}(\mathbf{w}^{(k)})\,.$$

W ten sposób otrzymaliśmy *kierunek* poprawki. Samej poprawki $\Delta\mathbf{w}^{(k)}$ dokonujemy korzystając z wzoru 5.3 lub 5.4.

Wadą metody jest niewykorzystywanie informacji o krzywiźnie funkcji (powierzchni błędu). Ponadto, w okolicy minimum gradient bywa mały, a więc zbliżanie się do minimum staje się bardzo powolne.

Zaletą metody jest jej prostota, małe wymagania co do pamięci, stosunkowo mała złożoność obliczeniowa.

Ponadto — można uzyskać poprawę efektywności metody przez zastosowanie momentum.

Dalsze modyfikacje tej metody to: reguła *delta-bar-delta* (wyznacza m.in. w specjalny sposób współczynnik uczenia η) i metoda *quickprop* Fahlmana.

5.3.4. Metoda Newtona

Korzystamy tutaj z przybliżenia kwadratowego funkcji $E(\mathbf{w})$:

$$E(\mathbf{w}^{(k)} + \mathbf{p}^{(k)}) \approx E(\mathbf{w}^{(k)}) + \mathbf{g}(\mathbf{w}^{(k)})^T \mathbf{p}^{(k)} + \frac{1}{2}(\mathbf{p}^{(k)})^T \mathbf{H}(\mathbf{w}^{(k)})\mathbf{p}^{(k)}. \tag{5.9}$$

Szukamy takiego wektora $\mathbf{p}^{(k)}$, który by — dodany do wektora $\mathbf{w}^{(k)}$ — pozwolił nam osiągnąć minimum powierzchni błędu.

Jeżeli osiągnęliśmy minimum, to pochodna w tym punkcie powinna być równa zero:

$$\frac{\partial E(\mathbf{w}^{(k)} + \mathbf{p}^{(k)})}{\partial \mathbf{p}} = 0.$$

Różniczkując obustronnie równość (5.9) ze względu na wektor $\mathbf{p}^{(k)}$, i korzystając z tego, że pochodna lewej strony wyznaczana w minimum jest równa zeru, otrzymujemy:

$$0 = \mathbf{g}(\mathbf{w}^{(k)}) + \mathbf{H}(\mathbf{w}^{(k)})\mathbf{p}^{(k)}.$$

Wynika stąd, że jako poszukiwany wektor kierunkowy $\mathbf{p}^{(k)}$ doprowadzający nas do minimum należy przyjąć

$$\mathbf{p}^{(k)} = -[\mathbf{H}(\mathbf{w}^{(k)})]^{-1}\mathbf{g}(\mathbf{w}^{(k)})\,. \qquad (5.10)$$

Aby zapewnić ten warunek, należy w każdym cyklu określić wartość gradientu $\mathbf{g}$ oraz Hesjanu $\mathbf{H}$ w punkcie znanego (ostatniego) rozwiązania $\mathbf{w}^{(k)}$.

Wtedy:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - [\mathbf{H}(\mathbf{w}^{(k)})]^{-1}\mathbf{g}(\mathbf{w}^{(k)})\,. \qquad (5.11)$$

Wzór ten jest trudny do zrealizowania w praktyce, gdyż

- wymaga w każdym kroku iteracyjnym (k) obliczenia wartości hesjanu $\mathbf{H}$ w punkcie $\mathbf{w}^{(k)}$,
- wymaga odwrócenia tego hesjanu, do czego jest potrzebna dodatnia określoność tego hesjanu — i to w każdym kroku iteracji — czego sam algorytm nie jest nam w stanie zapewnić.

W tym stanie rzeczy w praktycznych implementacjach rezygnuje się z wyznaczania dokładnej wartości hesjanu, a zamiast tego korzysta się z jego wartości przybliżonej, która w dalszym ciągu będzie oznaczana jako $\hat{\mathbf{H}}(\mathbf{w}^{(k)})$.

Jedną z najpopularniejszych metod służących temu celowi jest metoda zmiennej metryki.

5.3.5. Metoda zmiennej metryki — quasi-Newton

W metodzie tej w każdym kroku iteracji modyfikuje się hesjan $\hat{\mathbf{H}}$ z kroku poprzedniego o pewną poprawkę.

Poprawka może być tak dobrana, aby aktualna wartość hesjanu $\hat{\mathbf{H}}(\mathbf{w}^{(k)})$ przybliżała krzywiznę funkcji celu E zgodnie z zależnością:

$$\hat{\mathbf{H}}(\mathbf{w}^{(k)})(\mathbf{w}^{(k)} - \mathbf{w}^{(k-1)}) = \mathbf{g}(\mathbf{w}^{(k)}) - \mathbf{g}(\mathbf{w}^{(k-1)}),$$

gdzie $\mathbf{g}(\cdot)$ oznacza odpowiedni gradient.

Na podstawie powyższego założenia można otrzymać wzory określające hesjan w kroku k-tym w zależności od wartości hesjanu w kroku $(k-1)$-szym oraz od przyrostów gradientu i optymizowanej zmiennej $\mathbf{w}$. Przybliżenie hesjanu $\hat{\mathbf{H}}(\mathbf{w}^{(k)})$ powinno być tak skonstruowane, aby zapewnić istnienie odwrotności $\mathbf{V} = [\hat{\mathbf{H}}(\mathbf{w}^{(k)})]^{-1}$.

Osowski [12] (str. 57) podaje dwa algorytmy posiadające tę własność. Są to:
- algorytm Broydena-Goldfarba-Fletchera-Shanno (BFGS) oraz
- algorytm Dawidona-Fletchera-Powella (DFP).

Metoda zmiennej metryki charakteryzuje się zbieżnością superliniową; jest więc znacznie lepsza od liniowo zbieżnej metody największego spadku.

Według Osowskiego (str. 57) metoda ta jest obecnie uważana za jedną z najlepszych metod optymalizacji funkcji wielu zmiennych.

Jej wadą jest stosunkowo duża złożoność obliczeniowa wynikająca z konieczności wyznaczania S^2 elementów hesjanu, a także duże wymagania co do pamięci przy przechowywaniu macierzy hesjanu. Jej skuteczność sprawdzono na komputerach osobistych dla sieci nie przekraczających tysiąca parametrów.

5.3.6. Algorytm Levenberga-Marquardta

Jest inną, bardzo popularną odmianą newtonowskiej metody optymalizacji. Wartość dokładna hesjanu ze wzoru (5.10) zostaje zastąpiona wartością aproksymowaną $\hat{\mathbf{H}}(\mathbf{w}^{(k)})$ określaną na podstawie informacji zawartej w gradiencie, a także przy uwzględnieniu czynnika regularyzującego (Osowski, str. 58–59).

5.3.7. Metoda sprzężonych gradientów

Można by usiłować budować ciąg kolejnych kierunków $\mathbf{p}^{(k)}$ określających kolejne poprawki $\Delta\mathbf{w}^{(k)}$ (por. wzór (5.2)) jako ciąg wektorów $\mathbf{p}^{(k)}$ wzajemnie ortogonalnych. Zapewnia to przeszukiwanie przestrzeni optymalizowanych parametrów w niezależnych kierunkach. Jednak w takim przypadku nie wykorzystujemy informacji o krzywiźnie powierzchni błędu. Aby to zrobić, konstruujemy ciąg kolejnych przybliżeń, czyli ciąg wektorów $\mathbf{p}^{(k)}$ w ten sposób, żeby one były wzajemnie sprzężone względem hesjanu $\mathbf{H}$ lub jego przybliżenia.

Definicja. Wektory $\mathbf{p}_j$ i $\mathbf{p}_i$ są wzajemnie sprzężone względem macierzy $\mathbf{G}$, jeśli jest spełniony warunek:

$$\mathbf{p}_j^T \mathbf{G} \mathbf{p}_i = 0.$$

Osowski (str. 59) pokazuje, że kolejny kierunek $\mathbf{p}^{(k)}$ można otrzymać jako kombinację liniową aktualnego gradientu $\mathbf{g}^{(k)}$ i kierunków $\mathbf{p}^{(0)}, \ldots, \mathbf{p}^{(k-1)}$ wyznaczonych w poprzednich iteracjach:

$$\mathbf{p}^{(k)} = -\mathbf{g}_k + \sum_{j=0}^{k-1} \beta_{kj} \mathbf{p}^{(j)},$$

gdzie $\mathbf{g}_k$ jest gradientem funkcji błędu wyznaczonym w punkcie $\mathbf{w}_k$.

Po uwzględnieniu warunków ortogonalności i warunków sprzężenia względem hesjanu $\mathbf{H}$ otrzymujemy

$$\mathbf{p}^{(k)} = -\mathbf{g}_k + \beta_{k-1} \mathbf{p}^{(k-1)},$$

gdzie współczynnik β_{k-1} może być wyznaczony np. według jednej z następujących reguł (Osowski, str. 60, Bishop, str. 280–281):

$$\beta_{k-1} = \frac{\mathbf{g}_k^T (\mathbf{g}_k - \mathbf{g}_{k-1})}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \qquad \text{(Polak-Ribiere)}$$

$$\beta_{k-1} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{-(\mathbf{p}^{k-1})^T\mathbf{g}_{k-1}}$$

$$\beta_{k-1} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{(\mathbf{p}^{k-1})^T(\mathbf{g}_k - \mathbf{g}_{k-1})} \qquad \text{(Hestenes-Stiefel)}$$

$$\beta_{k-1} = \frac{\mathbf{g}_k^T\mathbf{g}_k}{\mathbf{g}_{k-1}^T\mathbf{g}_{k-1}} \qquad \text{(Fletcher-Reeves)}$$

Tak więc — korzystając z zasady sprzężenia — możemy konstruować kierunki poszukiwań.

Następnym etapem jest *znalezienie minimum funkcji na danym kierunku* $\mathbf{p}^k$ — przez tzw. minimalizację kierunkową. Wykorzystujemy tu aproksymację kwadratową wynikającą z rozwinięcia Taylora (5.9) postaci:

$$E(\mathbf{w}^{(k)} + \alpha\mathbf{p}^{(k)}) \approx E(\mathbf{w}^{(k)}) + \mathbf{g_k}^T(\mathbf{w}^{(k)} + \alpha\mathbf{p}^{(k)}) + \frac{1}{2}(\mathbf{w}^{(k)} + \alpha\mathbf{p}^{(k)})^T\mathbf{H}(\mathbf{w}^{(k)} + \alpha\mathbf{p}^{(k)}).$$

Teraz szukamy takiej wartości α która dałaby minimum funkcji błędu na kierunku $(\mathbf{w}^{(k)} + \alpha\mathbf{p}^{(k)})$.

Obliczając pochodną cząstkową względem α i przyrównując ją do zera otrzymamy

$$\alpha_{min}^{(k)} = \frac{\mathbf{p}^{(k)}\mathbf{g}_k}{(\mathbf{p}^{(k)})^T\mathbf{H}\mathbf{p}^{(k)}}. \tag{5.12}$$

Jak można zauważyć, do wyznaczenia wielkości $\alpha_{min}^{(k)}$ wystarczy znajomość $\mathbf{H}\mathbf{p}^{(k)}$ — który można otrzymać pośrednio, bez oddzielnego wyznaczania całego hesjanu $\mathbf{H}$.

Typowo, obydwa te kroki (wyznaczanie nowego, sprzężonego kierunku $\mathbf{p}^{(k)}$ i szukanie stałej α_k dającej minimum na tym kierunku) są stosowane naprzemiennie.

Własności metody. Według Osowskiego (str. 60) metoda sprzężonych gradientów wykazuje zbieżność zbliżoną do liniowej i z tego powodu jest mniej skuteczna niż metoda zmiennej metryki, ale zdecydowanie szybsza niż metoda największego spadku. Stosuje się ją powszechnie jako jedyny skuteczny sposób algorytmu optymalizacji przy bardzo dużej liczbie zmiennych sięgających nawet kilkudziesięciu tysięcy.

5.3.8. Metoda gradientów sprzężonych z regularyzacją (scaled conjugate gradients)

Algorytm *scg* jest dość skomplikowany. Jest on opisany u Osowskiego na str. 68–71 i Bishopa str. 282–285, jednak opisy w obu tych źródłach różnią się.

Generalnie, algorytm ten wiąże się z nazwiskiem Mollera i pozwala unikać bezpośredniego szukania minimum na danym kierunku przez tzw. line search. Algorytm

stosuje regularyzację dla uniknięcia trudności wynikających z faktu, że hesjan rozważany w tej metodzie (wynikający z aproksymacji kwadratowej funkcji błędu w punkcie $\mathbf{w}^{(k)}$) na ogół bywa niedodatnio określony lub źle uwarunkowany.

Regularyzacja jakiejś wielkości wiąże się generalnie z dodaniem pewnego warunku na tę wielkość. Z dodaniem nowego warunku wiąże się dodanie nowego parametru do rozpatrywanego modelu.

W omawianym zagadnieniu regularyzacja może zawierać dwa momenty:

i) uczynić hesjan dodatnio określony przez dodanie do macierzy $\mathbf{H}$ pewnej krotności λ macierzy jednostkowej; tym samym w dalszym ciągu będzie rozpatrywana macierz $\mathbf{H} + \lambda\mathbf{I}$;

ii) wprowadzić regularyzację do formuły na długość kroku $\alpha^{(k)}$ wyznaczanego według wzoru (5.12), co powoduje, że krok ten będzie wyznaczany z następującego warunku

$$\alpha_{min}^{(k)} = \frac{\mathbf{p}^{(k)}\mathbf{g_k}}{(\mathbf{p}^{(k)})^T\mathbf{H}\mathbf{p}^{(k)} + \lambda_k \left\|\mathbf{p}^{(k)}\right\|^2}. \tag{5.13}$$

Parametr λ_k występujący w warunku regularyzacji jest dobierany indywidualnie w każdym kroku iteracji.

Jak pisze Bishop (str. 285), rezultaty symulacji komputerowych wykazują, że w niektórych przypadkach algorytm *scg* może dać znaczącą poprawę w szybkości obliczeń w stosunku do zwykłego algorytmu sprzężonych gradientów.

Rozdział 6

Sieci o radialnych funkcjach bazowych czyli sieci typu RBF

Sieci te są opisane np. w książkach Osowskiego ([12] rozdział 5) i Bishopa ([1] również rozdział 5).

6.1. Architektura i funkcjonowanie sieci typu RBF

Sieci tego typu służą najczęściej do aproksymacji zmiennej numerycznej. Są używane również w zagadnieniach klasyfikacyjnych. Z różnych twierdzeń (np. Poggio i Girosiego) wynika, że taka sieć może służyć jako uniwersalny aproksymator i przybliżyć dowolną funkcję ciągła z dowolną dokładnością.

Sieci RBF są sieciami dwuwarstwowymi, z przepływem informacji do przodu, bez wymiany informacji między neuronami w jednej warstwie.

Tak więc sieć typu RBF oprócz warstwy wejściowej dostarczającej dane $\mathbf{x}$ składa się:

1. z warstwy H neuronów ukrytych,
2. z warstwy wyjściowej liczącej ogólnie K wyjść.

W poprzednio omawianych sieciach jedno- i wielowarstwowych neurony występujące w liczących się warstwach spełniały rolę sumatorów impulsów (bodźców) dochodzących do nich z poprzedniej warstwy. Rezultat sumowania był następnie przetwarzany przez tzw. funkcję aktywacji i powodował pobudzenie neuronu na określonym poziomie, co z kolei miało ten skutek, że dany neuron przekazywał swoją aktywację — jako bodziec numeryczny — do następnej warstwy, lub na wyjście.

W przypadku sieci typu RBF każdy neuron warstwy ukrytej odgrywa rolę małego kalkulatorka: wyznaczanie aktywacji odbywa się tu na zasadzie wyznaczenia wartości tzw. *radialnej funkcji bazowej* — o której zaraz powiemy coś więcej. Poza tym, wszystko odbywa się tak samo, jak w zwykłych sieciach dwuwarstwowych.

Definicja. Radialną funkcją bazową (typu RBF) nazywamy funkcję $G(\cdot)$ postaci

$$G(\mathbf{x};\mathbf{c}) = G(\,r(\mathbf{x},\mathbf{c}\,)), \quad \text{gdzie} \;\; r(\mathbf{x},\mathbf{c}\,) = \|\mathbf{x}-\mathbf{c}\|_2^2 = \{(\mathbf{x}-\mathbf{c})^T(\mathbf{x}-\mathbf{c})\}^{1/2}. \tag{6.1}$$

Oznacza to, że wartości funkcji — dla danego argumentu $\mathbf{x}$ — zależą tylko od *odległości* jej argumentu $\mathbf{x}$ od centrum $\mathbf{c}$ będącym parametrem tej funkcji. Czasami taka pojedyncza funkcja radialna jest nazywana *jądrem* (kernel), a parametr σ *szerokością* (width) jądra.

Stosunkowo często używaną funkcją radialną jest funkcja Gaussa

$$G(\mathbf{x};\mathbf{c},\sigma^2) = \exp\{\frac{\|\mathbf{x}-\mathbf{c}\|^2}{2\sigma^2}\}. \tag{6.2}$$

Jest to funkcja o kształcie dzwonu. Szerokość dzwonu reguluje parametr σ^2 nazywany również parametrem gładkości.

Przykłady innych funkcji radialnych (wraz z wykresami) można znaleźć np. w książce Osowskiego, str. 168, Rys. 5.4.

Tak więc neurony pierwszej warstwy obliczają — na podstawie podanego na wejściu wektora $\mathbf{x}$ — swoje aktywacje jako wartości

$$G_1(\mathbf{x}) = G(\mathbf{x},\mathbf{c}_1),\; G_2(\mathbf{x}) = G(\mathbf{x},\mathbf{c}_2),\; \dots\,,\; G_H(\mathbf{x}) = G(\mathbf{x},\mathbf{c}_H).$$

Subskrypt u dołu symbolu G oznacza tutaj numer centrum. Tak więc symbol G_h oznacza funkcję radialną obliczaną względem centrum $\mathbf{c}_h$.

Obliczone w ten sposób wartości $G_1, G_2, \dots, G_H$ służą jako dane wejściowe dla warstwy drugiej, wyjściowej, która oblicza z nich ważoną sumę ($k = 1, \dots, K$, gdzie K oznacza liczbę wyjść):

$$y_k(\mathbf{x}) = w_{k0} + w_{k1}G_1(\mathbf{x}) + w_{k2}G_2(\mathbf{x}) + \cdots + w_{kH}G_H(\mathbf{x}). \tag{6.3}$$

co można zapisać w postaci skróconej:

$$y_k = w_{k0} + w_{k1}G_1 + w_{k2}G_2 + \cdots + w_{kH}G_H. \tag{6.4}$$

Przy danych centrach $\mathbf{c}_1, \dots, \mathbf{c}_H$ i parametrze gładkości σ^2 pozostają nam do wyznaczenia tylko wektory wag $\mathbf{w}_1, \dots, \mathbf{w}_K$ używane przez warstwę wyjściową.

Wektory te otrzymuje się jako rezultat uczenia sieci za pomocą tzw. próbki uczącej. Próbka taka zawiera zbiór wzorców danych przez pary $\{\mathbf{x}^n, \mathbf{t}^n\}$, dla których przy każdym wektorze $\mathbf{x}^n$ znana jest prawidłowa odpowiedź $\mathbf{t}^n$, jakiej sieć powinna dostarczyć.

W dalszym ciągu omówimy bardziej szczegółowo tylko przypadek sieci typu RBF z jednym wyjściem.

6.2. Wyznaczanie centrów i parametrów gładkości

Centra — w zadeklarowanej liczbie H — są rozstawiane w przestrzeni zmiennych objaśniających R^d. Wyznaczanie centrów może się odbywać w zasadzie na trzy sposoby:

1. w sposób losowy;
2. przez podział danych punktów (określonych zmiennymi objaśniającymi) na H klasterów;
3. zestartowanie z liczbą klasterów równą liczebności próbki uczącej — każdy punkt tej próbki stanowi wtedy centrum; a następnie przez stopniową redukcję liczby centrów aż zostanie ich tylko H.

Parametr σ^2 wyznacza się na ogół metodą cross-validation (k-fold cross-validation).

Możliwy jest również algorytm który przyjmuje, że punkty próbki uczącej są mieszaniną rozkładów gaussowskich o indywidualnych centrach i indywidualnych szerokościach jąder. Jednoczesne wyznaczanie centrów $\mathbf{c}_h, \quad (h = 1, \dots, H)$ i odpowiadających im szerokości σ_h jest możliwe dzięki naprzemiennemu zastosowaniu algorytmu EM. Algorytm takiego wyznaczania klasterów jest zrealizowany w pakiecie Netlab.

6.3. Sieć typu RBF z jednym wyjściem i jej uczenie

Zakładamy, że w warstwie ukrytej znajduje się H neuronów, z których każdy (np. neuron h-ty) ma przyporządkowane swoje indywidualne centrum $\mathbf{c}_h$ określające strefę wpływów tego neuronu.

Zakładamy również, że aktywacja każdego neuronu będzie się odbywać za pośrednictwem radialnej funkcji Gaussa o parametrze σ^2 wspólnym dla wszystkich neuronów.

Tak więc mamy następującą sytuację

$$\begin{array}{cc} \text{Neuron} & \text{Funkcja aktywacji} \\ \text{Neuron 1 :} & G(\|\mathbf{x} - \mathbf{c}_1\|^2, \sigma^2) \\ \text{Neuron 2 :} & G(\|\mathbf{x} - \mathbf{c}_2\|^2, \sigma^2) \\ \cdots & \cdots \\ \text{Neuron H :} & G(\|\mathbf{x} - \mathbf{c}_H\|^2, \sigma^2) \end{array}$$

Możliwe są również modele, w których każdy neuron posiada swój własny parametr σ^2.

6.3.1. Uczenie sieci

Ponieważ mamy tylko jedno wyjście, więc zbiór próbek uczących (wzorców) będzie miał postać

$$\{\mathbf{x}^n, t^n\}, \quad n = 1, \dots, N.$$

Tym samym, dla n-tego wzorca warstwa ukryta dostarcza wartości

$$G_1(\mathbf{x}^n),\ G_2(\mathbf{x}^n),\ \dots\ , G_H(\mathbf{x}^n),$$

z których jest obliczany wynik (odpowiedź) sieci na sygnał $\mathbf{x}^n$:

$$y(\mathbf{x}^n) = w_1 G_1 + w_2 G_2 + \dots\ w_H G_h + w_0.$$

Porównując odpowiedzi sieci $y(\mathbf{x}^n)$ z wartościami docelowymi t^n określamy ogólny błąd odpowiedzi E. Błąd ten jest wyznaczany najczęściej na zasadzie najmniejszych kwadratów:

$$E = \frac{1}{2}\sum_{n=1}^{N}[t^n - y(\mathbf{x}^n)]^2 = \frac{1}{2}\sum_{n=1}^{N}[t^n - \sum_{j=0}^{H} w_j G_j(\mathbf{x}^n)]^2, \qquad (6.5)$$

gdzie symbolem $G_0 = G_0(\mathbf{x}^n)$ oznaczono funkcję przyjmującą tożsamościowo wartość równą 1.

Ponieważ wartości $G_j(\mathbf{x}^n)$ są znane, błąd E jest kwadratową funkcją wag $\mathbf{w} = [w_0, w_1, w_2, \dots, w_H]$. W takim przypadku (tj. gdy pod kwadratem występuje liniowa funkcja szukanych parametrów $w_0, w_1, \dots, w_H$), wektor wag $\mathbf{w}$ dający minimum błędu może być znaleziony explicite jako rozwiązanie układu równań liniowych (np. tzw. układu równań normalnych). **Nie potrzeba już metod iteracyjnych** — co czyni sieć typu RBF bardzo atrakcyjną z punktu widzenia obliczeniowego.

Jednak — przy rozwiązywaniu układu równań liniowych można również napotkać na trudności, np. otrzymany układ może być źle uwarunkowany.

W książce Osowskiego można znaleźć — w rozdziale 5 poświęconym sieciom RBF — kilka metod rozwiązywania układu równań liniowych wyznaczających wagi używane przez takie sieci. Może to być np. pseudoodwrotność lub SVD.

6.4. Implementacja sieci radialnych w pakiecie Netlab

Pakiet Netlab posiada swój własny konstruktor dla sieci RBF. Jego działanie pokazuje moduł demonstracyjny DEMRBF1. Moduł ten pokazuje, jak można otrzymać przybliżenie sinusoidy za pomocą różnego typu funkcji radialnych: gaussowskiej, thin plate spline ('TPS') ($G(r) = r^2 \log r$) oraz 'r4logr'.

Przed przystąpieniem do obliczeń dane powinny być wystandaryzowane.

A oto przykładowy skrypt wykonujący obliczenia dla 6-u różnych liczebności warstwy ukrytej.

```
% Demonstrates calculations of a~RBF approximation of a~sine function
% Generate the matrix of inputs x and targets t.
```

```
randn('state', 42);  rand('state', 42);
ndata = 20;          % Number of data points.
noise = 0.2;         % Standard deviation of noise distribution.
x = (linspace(0, 1, ndata))';
t = sin(2*pi*x) + noise*randn(ndata, 1);
mu = mean(x);  sigma = std(x);
tr_in = (x - mu) .// (sigma);  % Standardized data

H=[2 4 7 12 15 18]; % Number  of hidden neurons
for hh=H

     % Set up network parameters.
     nin = 1;                        % Number of inputs.
     nhidden = hh;                   % Number of hidden units.
     nout = 1;                       % Number of outputs.

     % Create and initialize network weight and parameter vectors.
     net = rbf(nin, nhidden, nout, 'gaussian');

     % Use fast training method
     options = foptions;
     options(1) = 0;         % No Display of EM when making clusters
     options(14) = 10;       % number of iterations of EM
     net = rbftrain(net, options, tr_in, t);
     y = rbffwd(net, inputvals);

     disp(['RBF training errors are  for ',...
           'nhidden H= ', num2str(nhidden),...
          ' RBF Gaussian ', num2str(rbferr(net, tr_in, t))]);

end
```

W rezultacie otrzymujemy następujące wartości błędu E:

```
-------------------------------------------------
RBF training errors are for nhidden H=2
Gaussian 0.5087 TPS 4.5182 R4logr 4.8817

RBF training errors are  for nhidden   H=4
Gaussian 0.24256 TPS 0.70335 R4logr 0.9769

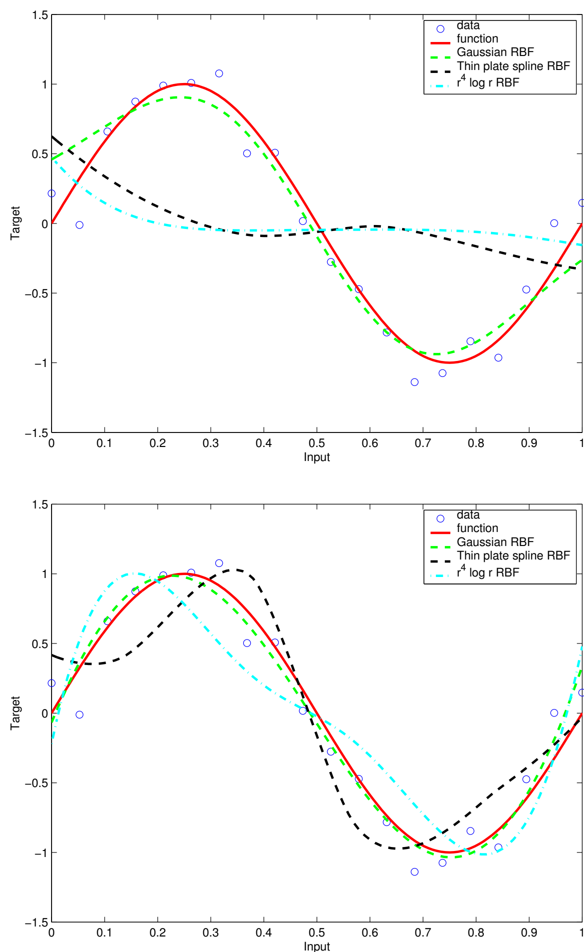
RBF training errors are  for nhidden  H=7
Gaussian 0.14586 TPS 0.14092 R4logr 0.26191

RBF training errors are  for nhidden  H=12
Gaussian 0.10797 TPS 0.089096 R4logr 0.097692
```

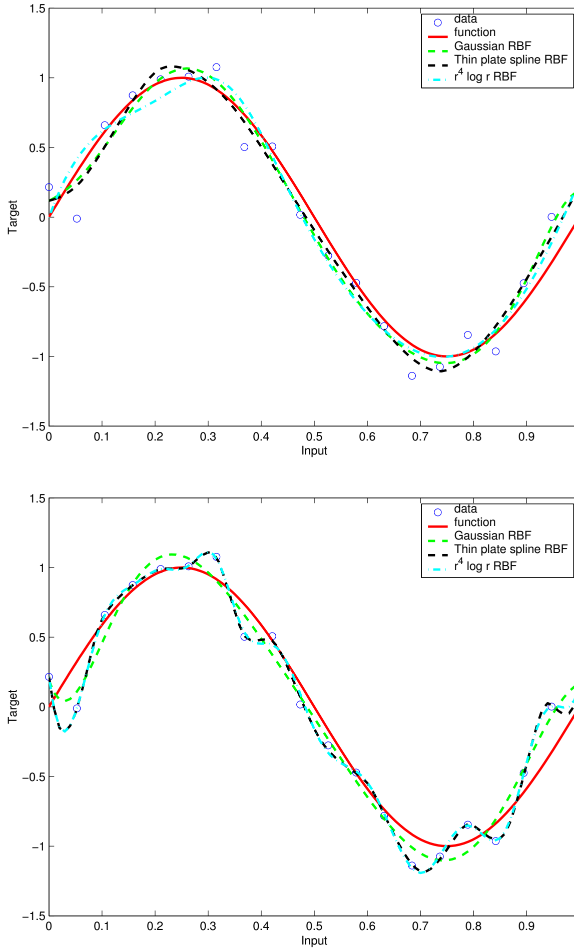
```
RBF training errors are  for nhidden  H=15
Gaussian 0.10797 TPS 0.0032335 R4logr 0.0073178

RBF training errors are  for nhidden  H=18
Gaussian 0.10792 TPS 0.0014216 R4logr 0.0043882
-------------------------------------------------
```

Wykonano również (za pomocą DEMRBF1) rysunki ilustrujące aproksymację funkcji sinus za pomocą sieci radialnych z H=2, 4, 7 i 15 neuronami.



Rysunek 6.1: *Aproksymacja funkcji sinus wykonana za pomocą sieci o H=2 (góra) i H=4 (dół) neuronach. Sieci zostały wytrenowane na podstawie 20-elementowej zaburzonej próbki wylosowanej z sinusoidy.*



Rysunek 6.2: *To samo, co na rysunku 6.1, ale warstwa ukryta sieci składała się z H=7 (góra) i H=15 (dół) neuronów.*

Rozdział 7

Uczenie samoorganizujące się

7.1. Uwagi Tadeusiewicza nt. Sieci samoorganizujących się

Sieci samoorganizujące się wykonują *grupowanie danych*. Pojawiają się dwie zasadnicze własności powodujące istotne zmiany w funkcjonowaniu sieci:

A. Koherencja,

B. Kolektywność.

Ad A.
Zasada **koherencji** postuluje, żeby grupować dane w pewne klasy podobieństwa. Grupowanie następuje na zasadzie samoorganizacji. Proces samoorganizacji jest automatyczny i spontaniczny, bez nauczyciela. Zakłada się, że komplet potrzebnych informacji jest zawarty w samych danych, z których jedne są podobne do siebie nawzajem, a inne nie.

Ad B.
Zasada **kolektywności** powoduje, że to, co rozpoznaje jeden neuron, zależy w dużej mierze od tego, co poznają inne neurony.

Zbiorowość neuronów (czyli ich kolektyw) może w sposób pełniejszy i bogatszy przetwarzać informacje, aniżeli każdy z neuronów wzięty z osobna.

Zbiorowość odpowiednio powiązanych i współpracujących elementów stwarza możliwość uzyskania nowych form zachowania i nowych postaci działań znacznie bogatszych, niż by można oczekiwać biorąc pod uwagę każdy z elementów tej zbiorowości z osobna. Przykłady z przyrody: Pszczoły, mrówki — jako kolektyw są zdolne do celowych działań.

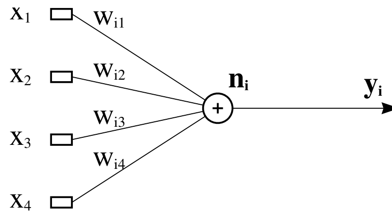
7.2. Zasada Hebba

7.2.1. Oznaczenia wstępne

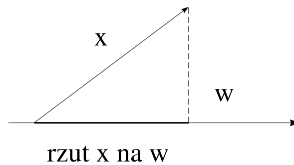
Rozpatrujemy prosty element sieci neuronowej[1], w której neuron o numerze i pobiera w k-tym kroku (k-tej iteracji) dane $\mathbf{x}(k) = \mathbf{x} = [x_1, \ldots, x_d]^T$ i na tej podstawie, korzystając ze swoich aktualnych wag $\mathbf{w}_i = [w_{i1}, \ldots, w_{id}]^T$ wytwarza wynik y_i. Wynik ten jest obliczany jako

$$y_i = \mathbf{w}_i^T \mathbf{x} = \mathbf{x}^T \mathbf{w}_i.$$

Element sieci pokazujący jeden neuron, oznaczony numerem i, jest pokazany na rysunku 7.1. Zauważmy, że neuron ten posiada swój wektor wag, $\mathbf{w}_i$, na podstawie



Rysunek 7.1: *Model liniowego neuronu z wyjściem* $y_i = \mathbf{w}_i^T \mathbf{x}$.



Rysunek 7.2: *Rzutowanie wektora x na wektor w*

których oblicza swoją aktywację jako kombinację liniową $\sum_j x_j w_{ij}, \quad j = 1, 2, \ldots, d$, gdzie d jest wymiarem wektora wejściowego, czyli sygnału pre-synaptycznego; na rysunku wymiar ten wynosi $d = 4$. Otrzymane w ten sposób pobudzenie neuronu jest kierowane na wyjście jako tzw. sygnał post-synaptyczny y_i. Tak więc, tak naprawdę, wartość wyjścia y_i zależy od wielkości sygnału pre-synaptycznego $\mathbf{x}$ oraz wektora wagowego $\mathbf{w_i} = [w_{i1}, \ldots, w_{id}]^T$ przypisanego do neuronu n_i.

Zauważmy również, że gdyby wektor $\mathbf{w}_i$ był wektorem jednostkowym, to wartość y_i mogłaby być interpretowana jako rzut wektora $\mathbf{x}$ na wektor $\mathbf{w}_i$, co ilustruje rysunek

[1]por. Osowski [12], str. 26

7.2. Jeżeli wektor nie jest jednostkowy, to wartość y_i jest proporcjonalna do wielkości tego rzutu, albowiem:

$$\mathbf{w}_j^T \mathbf{x} = \sum_j w_{ij} x_j = \|\mathbf{w}_i\| \cdot \underbrace{\|x\| \cdot \cos\alpha} = \|\mathbf{w}_i\| \cdot y_i,$$

skąd

$$y_i = \frac{\sum_j w_{ij} x_j}{\|\mathbf{w}_i\|}.$$

7.2.2. Sformułowanie zasady Hebba

W neuro-fizjologii zauważono, że waga powiązań między dwoma neuronami wzrasta przy jednoczesnym pobudzeniu obu neuronów, w przeciwnym przypadku maleje.

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \Delta\mathbf{w}_i(k),$$

gdzie ogólnie

$$\Delta\mathbf{w}_i(k) = F(\mathbf{x}(k), y_i(k)).$$

Funkcja F jest funkcją sygnału wejściowego $\mathbf{x}(k)$, nazywanego również sygnałem pre-synaptycznym, a y_i oznacza sygnał wyjściowy (pojawiający się na i-tym Wyjściu) nazywany również sygnałem post-synaptycznym.

W klasycznym ujęciu Hebba funkcja F upraszcza się do funkcji iloczynowej. Przy uczeniu bez nauczyciela funkcja F przyjmuje wtedy postać:

$$\Delta\mathbf{w}_i(k) = \eta \cdot \mathbf{x}(k) \cdot y_i(k)\,, \tag{7.1}$$

natomiast przy uczeniu z nauczycielem w powyższym wzorze zamiast sygnału post-synaptycznego $y_i(k)$ należy podstawić wartość docelową (target) $t_i(k)$. Formuła (7.1) nosi nazwę *prostej reguły Hebba.*

7.2.3. Reguła Oji

Przy użyciu funkcji iloczynowej użytej w prostej regule Hebba wagi wzrastają wykładniczo z postępem uczenia i wielokrotną prezentacją tego samego wzorca, co jest zjawiskiem bardzo niepożądanym.

Aby temu zapobiec, wprowadzono reguły ograniczające przyrosty wag. Jedną taką modyfikacją jest reguła zapominania. Inną ważną modyfikację wprowadził Oja, który zaproponował, aby regułę Hebba stosować na sygnale pre-synaptycznym skorygowanym wg następującego wzoru:

$$\Delta\mathbf{w}_i(k) = \eta \cdot y_i(k)[\mathbf{x}(k) - y_i(k)\mathbf{w}_i(k)]. \tag{7.2}$$

Tutaj $\mathbf{x}$ jest zmniejszane (proporcjonalnie) o wielkość rzutu wektora $\mathbf{x}$ na wektor $\mathbf{w}_i$. Gdyby wektor $\mathbf{w}_i$ był jednostkowy, to byłby to dokładnie rzut (porównaj rysunek 7.2 i uwagi na ten temat w sekcji 7.2.1).

Oja pokazał, że przy uczeniu określonym wzorem (7.2) wektor $\mathbf{w}_i$ w trakcie uczenia stabilizuje się i przyjmuje wartość jednostkową.

Po wielokrotnej (kilka set lub kilka tysięcy) prezentacji próbek danych wartości *średnie* lub też *wartości oczekiwane* przyrostów wag powinny wynosić zero. Oznacza to, że po ustaleniu punktu równowagi

$$\mathcal{E}\{\Delta\mathbf{w}_i(k)\} = 0,$$

i tym samym również

$$\mathcal{E}\{\eta \cdot y_i(k)[\mathbf{x}(k) - y_i(k)\mathbf{w}_i(k)]\} = 0.$$

Opuszczając współczynnik η oraz wskaźniki (k) pokazujące zależności poszczególnych wielkości od numeru iteracji k, otrzymujemy:

$$\mathcal{E}\{\mathbf{x}y_i\} - \mathcal{E}\{y_i y_i \mathbf{w}_i\} = 0.$$

Biorąc pod uwagę, że $y_i = \mathbf{w}_i^T\mathbf{x} = \mathbf{x}^T\mathbf{w}_i$, otrzymujemy w dalszym ciągu:

$$\mathcal{E}\{\mathbf{x}\mathbf{x}^T\mathbf{w}_i\} - \mathcal{E}\{\underbrace{\mathbf{w}_i^T \underbrace{\mathbf{x}\mathbf{x}^T} \mathbf{w}_i}\, \mathbf{w}_i\} = 0.$$

Oznaczmy

$$\mathcal{E}\{\mathbf{x}\mathbf{x}^T\} = \mathbf{C}, \quad \text{oraz} \quad \mathbf{w}_i^T\mathbf{C}\mathbf{w}_i = \lambda_i\,. \tag{7.3}$$

W zależności od przeskalowania danych macierz $\mathbf{C}$ jest nazywana macierzą iloczynów skalarnych, macierzą kowariancji lub macierzą korelacji (danych). Po wykonaniu podstawienia określającego macierz $\mathbf{C}$ i skalar λ_i otrzymujemy:

$$\mathbf{C}\mathbf{w}_i = \lambda_i\mathbf{w}_i. \tag{7.4}$$

Jak widać, wektor $\mathbf{w}_i$ w stanie równowagi przedstawia sobą wektor własny macierzy $\mathbf{C}$ związany z wartością własną λ tej macierzy. Można pokazać, że musi to być wektor o długości 1: mnożąc (7.4) lewostronnie przez $\mathbf{w}_i^T$ i podstawiając $\mathbf{w}_i^T\mathbf{w}_i = \|\mathbf{w}_i\|^2$ otrzymujemy $\lambda_i = \lambda_i\|\mathbf{w}_i\|^2$, co jest prawdziwe tylko wtedy, gdy $\|\mathbf{w}_i\| = 1$.

Z innego rozumowania (nieprzedstawionego tutaj) wynika, że wektor $\mathbf{w}_i$ wskazuje na tak zwany kierunek główny zbioru danych, czyli oś główną elipsy (elipsoidy) przedstawiającą koncentrację punktów w przestrzeni danych.

Oja, a później Sanger, podali efektywne wzory, jak znajdywać jednocześnie wektory $\mathbf{w}_1, \ldots, \mathbf{w}_k$ przedstawiające tzw., kierunki główne danych. Technika ta ma duże zastosowanie przy redukcji liczby cech w danych: wektory $\mathbf{x} = [x_1, \ldots, x_d]^T$ o dużej

liczbie wymiarów można wtedy zastąpić wektorami $\mathbf{z} = [z_1, \ldots, z_k]^T$, $(k < d)$ co na ogół znacznie mniejszej liczbie cech.

Osowski (str. 235) pokazuje przykład zastosowania tej techniki do zapamiętywania i odtwarzania obrazów; pokazuje m.in. obraz odtworzony na podstawie dwóch pierwszych składowych głównych.

7.3. Uczenie z konkurencją

Neurony otrzymują na początku przypadkowe wartości wag, a następnie, na podstawie prezentowanych danych „uczą się" rozpoznawać te dane i zbliżają się odpowiednio do obszarów zajmowanych przez te dane.

Po każdej prezentacji wzorca $\mathbf{x}(k)$ zwycięzcą zostaje tylko jeden neuron, najbliższy prezentowanemu wzorcowi (bliskość jest liczona w sensie ustalonej metryki).

Przypuśćmy, że w tej konkurencji zwyciężył neuron o numerze c. Ma on prawo uaktualnić swoje wagi według jednej z dwóch zasad: *WTM* – Winner Takes Most, i *WTA* – Winner Takes All.

- Jeżeli obowiązuje zasada WTA, to tylko neuron zwycięski (oznaczany dalej indeksem c, od *conqueror*) uaktualnia swe wagi, tzn. zbliża się do wektora $\mathbf{x}$(k). Wagi zwycięskiego neuronu mogą zostać uaktualnione np. na podstawie wzoru Kohonena:
$$\mathbf{w}_c(k+1) = \mathbf{w}_c(k) + \eta(k)[\mathbf{x}(k) - \mathbf{w}_c(k)]. \tag{7.5}$$
Współczynnik uczenia $\eta(k)$ jest na ogól malejącą funkcją numeru iteracji k.
- Jeżeli obowiązuje zasada WTM to również neurony sąsiadujące z neuronem zwycięskim, czyli należące do sąsiedztwa $\mathcal{N}_c(k)$, mogą to uczynić według zasady:
$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \eta(k)G(i,c)[\mathbf{x}(k) - \mathbf{w}_i(k)], \quad i \in \mathcal{N}_c(k)\,. \tag{7.6}$$
Funkcja G oznacza tu wpływ sąsiedztwa i jest omawiana bliżej w rozdziale 8.

W efekcie takiego współzawodnictwa następuje samoorganizacja procesu uczenia. Neurony dopasowują swoje wagi w ten sposób, że przy prezentacji grup wektorów wejściowych zbliżonych do siebie zwycięża zawsze ten sam neuron.

Neuron, poprzez zwycięstwo we współzawodnictwie rozpoznaje swoją kategorię.

Ogólnie można powiedzieć, że: Przy podaniu na wejście sieci wielu wektorów zbliżonych do siebie będzie zwyciężać ciągle ten sam neuron, w wyniku czego jego wagi będą odpowiadać uśrednionym wartościom wektorów wejściowych, dla których dany neuron był zwycięzcą.

Neurony niewygrywające nie zmieniają swoich wag. Mówi się, że pozostają one martwe.

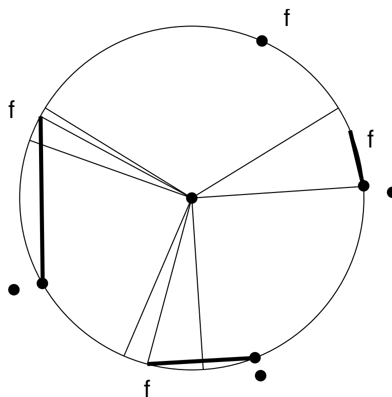
7.3.1. Przykład uczenia WTA wektorów leżących na kole

Zasadę uczenia z konkurencją WTA ilustruje rysunek 7.3 zaczerpnięty z książki Osowskiego. Zbiór danych składa się z ośmiu wektorów; każdy z tych wektorów jest dwuwymiarowy i ma długość jednostkową, wobec tego przestrzeń danych może być odwzorowana na okręgu (w przypadku wektorów danych wielowymiarowych o długości 1 – wektory danych mogłyby być odwzorowane na sferze).

Mamy również 4 neurony z przypisanymi do nich wektorami wagowymi $\mathbf{w_i} = [w_{i1}, w_{i2}]^T$, $i = 1, 2, 3, 4$, każdy z nich o długości jeden, tj. $\|\mathbf{w}_i\| = 1$, $\forall_i$.

Neurony te miały się, w procesie konkurencyjnym uczenia, nauczyć prezentowanego zbioru danych i zaadaptować swoje wagi w ten sposób, aby możliwie dobrze reprezentować te dane.

Uczenie było konkurencyjne (WTA), a elementy zbioru danych prezentowano neuronom w sumie 320 razy.



Rysunek 7.3: *Uczenie się z konkurencją. Jeden neuron nigdy nie zwyciężył, wskutek czego nie zdołał wytworzyć swojej strefy wpływów*

Po prezentacji kolejnego wzorca zwycięzcą zostawał neuron najbliższy temu wzorcowi (najbliższy w sensie odległości kątowej mierzonej po okręgu koła).

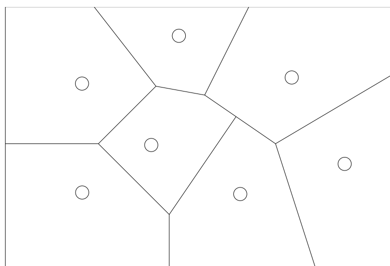
Po zakończeniu prezentacji okazało się, że w poszczególnych prezentacjach zwycięzcami zostawały tylko trzy neurony; i tylko one uzyskały przywilej zaadaptowania swoich wag. Jeden z neuronów nigdy nie zwyciężył, wobec czego jego wagi pozostały niezmienione. Pozostałe neurony uzyskały swoje strefy wpływów, W rezultacie neurony te uplasowały się w okolicy średnich pewnych grupek danych i stały się reprezentantami tych grupek danych.

7.3.2. Kwantyzacja przestrzeni danych i wieloboki Voronoia

Opisany w poprzedniej sekcji przykład uczenia się neuronów dla danych należących do R^2 może być zrealizowany dla danych należących do dowolnego zbioru punktów w R^d. W trakcie procesu uczenia neurony starają się uplasować pośród tych punktów–danych i stać się ich reprezentantami. W ten sposób cała przestrzeń zostaje podzielona na obszary atrakcji odzwierciedlające strefy wpływów poszczególnych neuronów. W przypadku używania metryki euklidesowej są to obszary wypukłe, które są nazywane są czasem wielobokami Voronoia, a reprezentujące je wektory — wektorami Voronoia. Obszary Voronoia V_i są zdefiniowane jako miejsce geometryczne punktów $\mathbf{x}$ spełniających następujący warunek:

$$V_i = \{\mathbf{x} : \ \|\mathbf{w}_i - \mathbf{x}\| < \|\mathbf{w}_j - \mathbf{x}\|, \ i \neq j\}$$

Przykład wieloboków Voronoia jest pokazany na rysunku 7.4 — gdzie mamy przestrzeń danych określoną jako prostokąt.



Rysunek 7.4: *Wieloboki Voronoia utworzone przez 7 wektorów kodowych na prostokącie z metryką euklidesową*

Podkreślmy tutaj, że używanie różnych metryk (tj. korzystanie z różnych definicji odległości między dwoma punktami leżącymi w przestrzeni danych) daje różne postaci obszarów atrakcji. Przykład rozkładu takich obszarów atrakcji — przy użyciu czterech różnych metryk — jest pokazany w książce Osowskiego, str. 252.

Kohonen nazwał proces tworzenia reprezentantów danych kwantowaniem wektorowym (*Vector Quantization*), lub dokładniej: adaptacyjnym kwantowaniem wektorowym (*LVQ, Learning Vector Quantization*). Wektory wagowe neuronów zostały przez Kohonena nazwane *słowami kodowymi, codebook vectors*, a ich zbiór — *książką kodową, codebook*

W ten sposób cała przestrzeń danych została skwantowana; a wektory danych należące do jednego wieloboku Voronoia można zastąpić odpowiadającym mu słowem kodowym.

Osowski (str. 268–275) podaje następujące przykłady zastosowań sieci samoorganizujących się:

1. Kompresja obrazów
2. Wykrywanie typu uszkodzeń
3. Prognozowanie obciążeń systemu elektroenergetycznego

Rozdział 8

SOMy czyli samoorganizujące się mapy

Typowy przedstawiciel takich sieci: Sieci Kohonena[1].
SOM-y (Self Organizing Maps) realizują generalnie dwa zadania:

1. Wektorowej kwantyzacji (kompresji danych),
2. Odtwarzanie przestrzennej organizacji danych wejściowych

8.1. Oznaczenia

Niech $\mathbf{x} = [x_1, \ldots, x_d]^T$ oznacza d-wymiarowy wektor danych, tzw. próbkę lub wzorzec. Wektor $\mathbf{x}$ może być interpretowany jako punkt d-wymiarowej przestrzeni: $\mathbf{x} \in R^d$.

Załóżmy, że mamy m neuronów tworzących mapę. Neurony te mogą być ułożone w siatkę $(m1 \times m2)$ na płaszczyźnie, lub liniowo na odcinku $1 \times m$). Możliwe (ale znacznie rzadziej wykorzystywane) są siatki trójwymiarowe, lub rozłożone na cylindrze lub torusie.

Każdy neuron jest scharakteryzowany swoim wektorem wag (nazywanym również wektorem kodowym lub wektorem Voronoia). Tak więc mamy m wektorów wagowych $\mathbf{w}_1, \ldots, \mathbf{w}_m$, gdzie $\mathbf{w}_i = [w_{i1}, w_{i2}, \ldots, w_{id}]^T, \;\; i = 1, \ldots, m$, $\mathbf{w}_i \in R^d$.

Te wszystkie pojęcia występowały już wcześniej.

Teraz definiujemy dodatkowe **wektory referencyjne** $\mathbf{r}_1, \ldots, \mathbf{r}_m$ określające pozycje węzłów siatki. Wektory referencyjne $\mathbf{r}_i$, $(i = 1, \ldots, m)$, są ściśle przyporządkowane wektorom kodowym $\mathbf{w}_i$ należącym do przestrzeni danych wejściowych (input space) R^d.

[1]opisane np. w książce Osowskiego, str. 249–275

W dalszym ciągu będziemy zajmować się tylko mapami na płaszczyźnie, wobec tego $\mathbf{r}_i \in R^2, \ i = 1, \dots, m$.

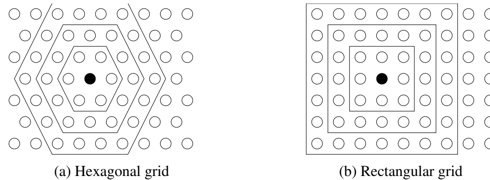
Inicjacja wektorów kodowych i przyporządkowań do nich odpowiednich wektorów referencyjnych jest na ogół przypadkowa. Sieć (mapa) na podstawie prezentowanych wzorców powinna nauczyć się danych. Uczenie jest konkurencyjne.

Jak wiemy, przy uczeniu konkurencyjnym zwycięża tylko jeden neuron (jego numer oznaczaliśmy indeksem c), ale neurony znajdujące się w sąsiedztwie neuronu–zwycięzcy mogą mogą też częściowo partycypować w zwycięstwie neuronu c i adaptować swoje wagi. Dlatego niezmiernie ważnym pojęciem jest pojęcie **sąsiedztwa** zwycięskiego neuronu.

8.2. Siatki i sąsiedztwa

Istotą sieci Kohonena jest ustalenie pewnej sztywnej siatki neuronów. Siatka taka (grid, map) ma najczęściej budowę hexagonalną lub prostokątną, jak to pokazano na rysunku 8.1.

Na siatce heksagonalnej każdy neuron ma sześć, a na prostokątnej osiem sąsiadów pierwszego rzędu. Można by liczyć również sąsiedztwo 2-go rzędu, i dalsze, jak to pokazano na rysunku 8.1.



Rysunek 8.1: *Siatka hexagonalna i prostokątna*

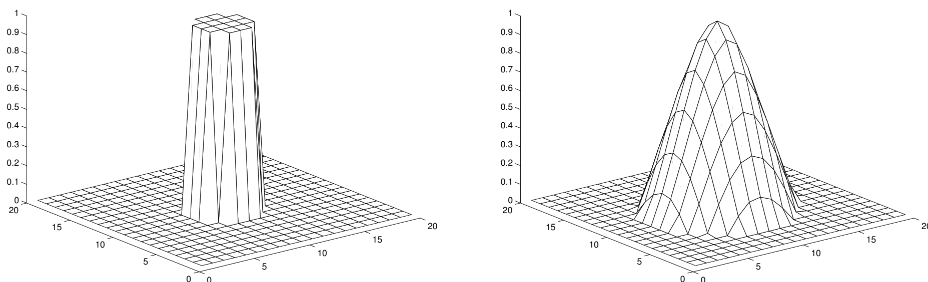
Sąsiedztwo można wyznaczać również analitycznie, za pomocą pewnych funkcji wyznaczających odległość neuronu i od zwycięskiego neuronu c. Funkcje te mają na ogół postać funkcji radialnych, zcentrowanych w wektorze referencyjnym $\mathbf{r}_c$.

Najbardziej popularnymi funkcjami sąsiedztwa są *bubble* i *gaussian*. Są one pokazane na rysunku 8.2. Przykładowo funkcja *gaussian* przyjmuje postać

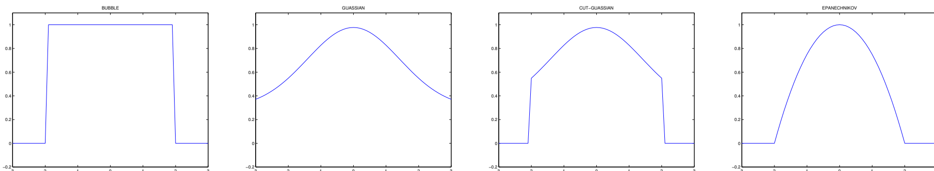
$$h_{gauss}(i, c) = exp\left(-\frac{\|\mathbf{r}_c - \mathbf{r}_i\|}{2\sigma^2(t)}\right)$$

Jeszcze inne funkcje sąsiedztwa (*cut-gaussian* i *Epanechnikov*) są pokazane na rysunku 8.3.

Sąsiedztwo neuronu c będziemy oznaczać $\mathcal{N}_c$ (od neighbourhood). Jeżeli chcemy wyraźnie napisać, że jest to sąsiedztwo neuronu c który zwyciężył w iteracji k, to



Rysunek 8.2: *Funkcje sąsiedztwa: bubble i gaussian*



Rysunek 8.3: *Jednowymiarowe funkcje sąsiedztwa dla promienia R=2: bubble, gaussian, cut-gaussian, Epanechnikov*

zapiszemy

$$\mathcal{N}_c(k).$$

Na ogół sąsiedztwo $\mathcal{N}_c(k)$ charakteryzuje się pewnym promieniem, który maleje wraz z upływem czasu uczenia t, czyli w miarę zwiększania się wskaźnika k, oznaczającego numer kolejny iteracji (prezentacji wzorców próbki uczącej).

8.3. Adaptacja wektorów kodowych podczas procesu uczenia

Proces uczenia polega na prezentowaniu sieci wzorców zbioru uczącego, z generalnym celem, żeby siec upodobniła swoje wektory kodowe do prezentowanych wzorców.

Podczas uczenia obowiązuje konkurencja.

Niech k będzie wskaźnikiem (numerem) kolejnej prezentacji. Po prezentacji sieci wektora danych $\mathbf{x}(k)$ zwycięża ten neuron, którego wektor kodowy jest najbliższy dopiero co zaprezentowanemu wektorowi $\mathbf{x}(k)$ w sensie ustalonej metryki (najczęściej jest to metryka euklidesowa).

Oznaczmy numer neuronu–zwycięzcy symbolem c (conqueror). Wektor kodowy

tego neuronu spełnia relację:

$$d(\mathbf{x}, \mathbf{w}_c) = \min_{1 \leqslant i \leqslant m} d(\mathbf{x}, \mathbf{w}_i).$$

Zwycięski neuron oraz neurony należące do jego sąsiedztwa (o ile aktualny zbiór $\mathcal{N}_c(k)$ obejmuje jakichś sąsiadów) mogą aktualizować swoje wagi, to znaczy upodabniać je do aktualnego zaprezentowanego wzorca $\mathbf{x}(k)$, natomiast wagi neuronów nie należących do $\mathcal{N}_c(k)$ nie zmieniają się.

8.4. Formuły na zmianę wag wygrywających neuronów

Jeżeli i-ty neuron otrzymał prawo zmiany swoich wag, to zmiana ta dokonuje się według wzoru:

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \eta(k) \cdot h_{ci}(k) \cdot [\mathbf{x}(k) - \mathbf{w}_i(k)],$$

gdzie $\eta(k)$ oznacza współczynnik uczenia — piszemy o nim w sekcji 8.6.
c oznacza numer wektora–zwycięzcy, tj. znajdującego się najbliżej prezentowanego w k-tym kroku wektora $\mathbf{x}(k)$,
$h_{ci}(k)$ określają, czy neuron i należy do sąsiedztwa zwycięskiego neuronu.

Jeśli chodzi o *współczynnik uczenia* $\eta_i(k)$, to na ogół maleje on wraz z upływem czasu uczenia wyznaczanego numerem iteracji k (por. sekcja 8.6);

Jeśli chodzi o *funkcję* $h_{ci}(k)$, to może ona być określona w następujący sposób:

$$h_{ci}(k) = \begin{cases} 1 & \text{gdy} \quad i \in N_c(k)\,, \\ 0 & \text{gdy} \quad i \notin N_c(k)\,. \end{cases}$$

Funkcja $h_{ci}(k)$ może też zależeć bezpośrednio od odległości D między wektorami referencyjnymi $\mathbf{r}_c$ i $\mathbf{r}_i$, np.

$$h_{ci}(k) = \begin{cases} \text{g(D(c,i))} & \text{gdy} \quad i \in N_c(k)\,, \\ 0 & \text{gdy} \quad i \notin N_c(k)\,, \end{cases}$$

gdzie $g(\cdot)$ jest funkcją malejącą (dokładniej: nierosnącą) przy wzroście swego argumentu. Może to być np. funkcja *bubble* lub *gaussian* pokazana na rysunku 8.2.

8.4.1. Dwie fazy uczenia

Na ogół uczenie przebiega w dwóch fazach. Najpierw przyjmuje się duże wartości η i duży promień sąsiedztwa.

W drugiej fazie (*fine tuning*) obydwie te wielkości ulegają istotnemu zmniejszeniu; w szczególności promień sąsiedztwa maleje do zera.

Pierwsza faza – przebiega według zasady WTM – promień sąsiedztwa jest duży, co powoduje, że oprócz neuronu–zwycięzcy również jego sąsiedzi z mapy zmieniają swoje wektory kodowe. Również współczynnik uczenia η jest stosunkowo duży.

Tak więc, przy każdej prezentacji kolejnego wektora $\mathbf{x}$ zostanie do niego przyciągnięty odpowiadający mu wektor–zwycięzca, który pociąga za sobą neurony z najbliższego sąsiedztwa na zdefiniowanej siatce.

Jeśli prezentujemy sieci np. wektory $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ położone blisko siebie w przestrzeni R^N, to znajdą one ten sam neuron w który będzie pociągał za sobą swoje otoczenie.

Druga faza uczenia. Obowiązuje tu zasada WTA. Adaptacji podlegają tylko neuron–zwycięzca c, ponieważ promień sąsiedztwa zmalał do zera. Wzór na adaptację:

$$\mathbf{w}_c(k+1) = \mathbf{w}_c(k) + \eta(k)[\mathbf{x} - \mathbf{w}_c(k)].$$

Ostatecznie cała przestrzeń R^d w której znajdują się dane, zostanie podzielona na strefy wpływów poszczególnych neuronów (obszary Voronoia). Sąsiedztwo wektory referencyjnych na mapie powinno odzwierciedlać (topologicznie) sąsiedztwo wektorów kodowych w przestrzeni danych.

8.5. Uczenie sieci on-line i wsadowo, współczynnik uczenia

Uczenie może się dokonywać na dwa sposoby: sekwencyjnie i wsadowo (batch).

Uczenie *sekwencyjne*, inaczej *na bieżąco*, lub *on–line* polega na tym, że dla $t = 1, 2, \ldots$ prezentujemy sieci wektory danych $\mathbf{x}(t)$, po czym następuje uaktualnienie wag zwycięskiego neuronu (i ewentualnie jego sąsiadów), jak to pokazano w sekcji 8.4.

Współczynnik uczenia na ogół zmniejsza się z czasem.

Niech T oznacza maksymalną liczbę iteracji. Stosuje się następujące wzory na zmniejszanie współczynnika uczenia:

1. Liniowe zmniejszanie
   $\eta(t) = \frac{\eta_0}{T}(T - t), \quad t = 1, 2, \ldots$.
2. Wykładnicze zmniejszanie
   $\eta(t) = \eta_0 exp(-Ct), \quad t = 1, 2, \ldots, \quad C > 0$ jest pewną stałą.
3. Hiperboliczne zmniejszanie
   $\eta(t) = \frac{C_1}{C_2 + t}, \quad t = 1, 2, \ldots, \quad C_1, C_2 > 0$ pewne stałe.
4. Indywidualny współczynnik uczenia, np.
   $\eta_i(t) = 1/n_i(t)$, gdzie $n_i(t)$ oznacza liczbę zwycięstw i-tego neuronu.

Wsadowe uczenie w przypadku SOM-ów jest znacznie szybsze i bardziej stabilne; polega na wykonywaniu aktualizacji wag na podstawie wszystkich próbek. Jest

to wariant domyślny w pakiecie somtoolbox vs.2. Algorytm uczenia wsadowego jest następujący (por. Skubalska, [14], za Kohonenem [7])

1. Ustal początkowe wektory kodowe i odpowiadające mu wektory referencyjne na mapie. Takimi początkowymi wektorami kodowymi może być np. m dowolnych wektorów danych z próbki uczącej.
2. Dla każdego wektora $\mathbf{w}_i$ zapamiętaj zbiór wektorów uczących $\{\mathbf{x}_k\}$ które oddziaływałyby na $\mathbf{w}_i$ w zwykłym algorytmie SOM.
3. Wyznacz nowe wartości wag (symbol $c(\mathbf{x}_k)$ oznacza neuron wygrywający przy prezentacji wektora danych $\mathbf{x}_k$, natomiast N jest ogólną liczebnością próbek uczących)
$$\mathbf{w}_i = \sum_{k=1}^{N} \mathbf{x}_k h_{c(\mathbf{x}_k)i} / \sum_{k=1}^{N} h_{c(\mathbf{x}_k)i}$$
4. Jeśli nie jest spełnione kryterium STOP-u (tutaj nie podaliśmy go), wróć do kroku 2.

8.6. Dobroć aproksymacji

Błąd kwantyzacji – jako średnia odległość punktów od ich reprezentantów czyli wektorów wag (data representation accuracy, average quantization error between data vectors and their BMU — best maching units — on the map).

Błąd topologicznej reprezentacji – procent wektorów danych, dla których pierwsi dwaj najbliżsi reprezentanci nie są sąsiadami na siatce (data set representation accuracy, the topographic error; percentage of data vectors for which the first- and second-BMU are not adjacent units).

8.7. Dostępne dla nas oprogramowanie

SOM_PAK – zestaw programów w języku C autorstwa Kohonena i jego zespołu. Jest dostępny w internecie, tworzy grafikę w postaci .ps lub .eps; wymaga kompilacji (make). Pracuje zarówno na Unix-ie jak i na PC-tach. Adres internetowy: `http://www.cis.hut.fi/research/som_lvq_pak`

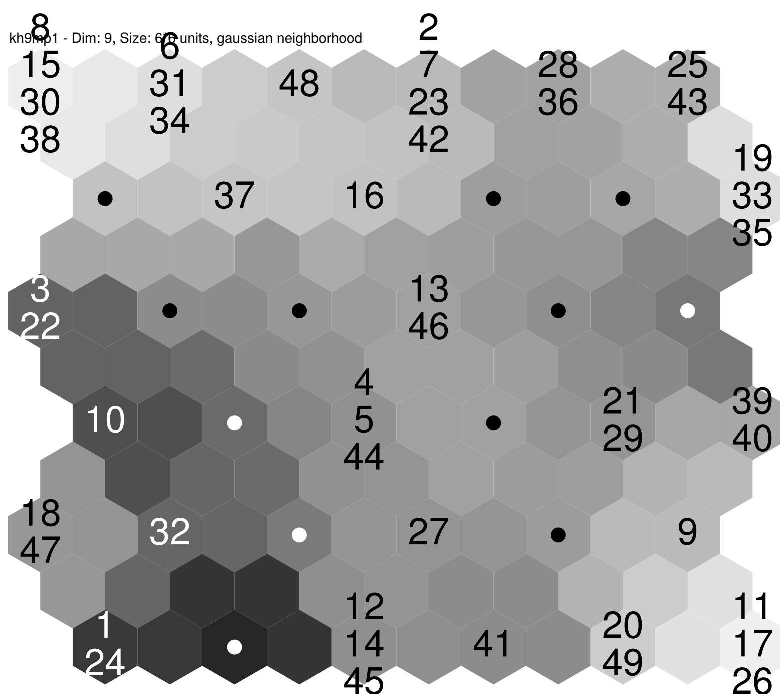
somtoolbox vs. 2 ([18]) – zestaw skryptów w postaci M-files autorstwa J. Vesanto i współautorów. Dostępny w internecie pod adresem : `http://www.cis.hut.fi/projects/somtoolbox`

8.8. Przykład tworzenia mapy dla 49 województw polskich; wizualizacja *U-mat*

Rozpatrujemy tablicę danych o wymiarze 49 × 9, której wiersze odpowiadają 49 województwom polskim (dane pochodzą z r. 1990), z których każde zostało scharakteryzowane przez 9 cech socjo-ekonomicznych (kolumny tablicy). Tym samym każde województwo może być interpretowane jako punkt w R^9.

Dla grupowania punktów zastosowano mapę heksagonalną o siatce 6 × 6.

Na rysunku 8.4 pokazujemy mapę otrzymaną w wyniku obliczeń oryginalnym programem SOM_PAK Kohonena (niestety, program ten nie liczy błędu topologicznej reprezentacji).



Rysunek 8.4: *Mapa Kohonena obrazująca zgrupowania 49 województw polskich ze względu na 9 cech socjo-ekonometrycznych. Niektóre węzły mapy pozostały puste, inne wektory kodowe zdołały przyciągnąć po kilka punktów–województw*

To co widzimy na siatce (mapie) jest obrazem wielowymiarowej przestrzeni. Siatka składa się z obszarów heksagonalnych, w których środkach znajdują się wektory referencyjne odpowiadające wektorom kodowym umiejscowionym w R^9. Oznacza to,

że każdemu neuronowi $\mathbf{r}_i$ na mapie odpowiada wektor kodowy (codebook vector) $\mathbf{w}_i$ w przestrzeni wejściowej (input space) R^9.

Faktyczne odległości między wektorami $\mathbf{w}_i$ są obrazowane odcieniami szarości na mapie: obszary bliskie są jasne, ciemny kolor oznacza duże odległości, a więc może oznaczać granice klasterów z R^9 (inne programy, np. Somtoolbox operują w tym celu kolorami z odpowiednim kluczem na oznaczenie bliskich i dalekich wektorów wagowych). Na utworzonej mapie właściwe heksagony (j) zawierające węzły mapy są otoczone dodatkowymi heksagonami pokazującymi kolorystycznie, jaka jest średnia odległość wektora kodowego (j) od sąsiadujących z nim wektorów kodowych.

Taki sposób wizualizacji nosi nazwę *u–mat* od informatyka o nazwisku Ultsch, który zaproponował taki sposób wizualizacji map.

Mając klucz do województw możemy próbować interpretować powstałe zgrupowania. Punkty 1 i 24 to województwa Warszawskie i Łódzkie. Punkty 18 i 47 to Kraków i Wrocław; punkt 32: Poznań; punkt 3 i 22 to Białystok i Lublin; punkt 10: Gdynia-Gdańsk.

Wszystkie te punkty to miasta uniwersyteckie z pewną tradycją. Tworzą one wyraźny klaster, oddzielony od pozostałych punktów zatoką pustych węzłów.

Innym widocznym na mapie klasterom można przypisać również ciekawą interpretację.

8.9. Inne algorytmy uczenia sieci samoorganizujących

Miary odległości między wektorami, problem normalizacji wektorów (należy dodać tzw. normalizację na 0–1, o czym Osowski nie mówi)

- Algorytmy uwzględniające „zmęczenie” neuronów po zwycięstwie.
- Algorytm stochastycznej relaksacji (neurony ulegają adaptacji z pb-stwem określonym wzorem Gibbsa).
- SCS, Soft Competition (uwzględnia się stopień aktywności neuronów, faworyzując jednostki mało aktywne).
  Współczynnik uczenia może być indywidualny dla każdego neuronu, jest uzależniony od tego, co robił neuron w ostatnich $k-1$ prezentacjach
- Algorytm gazu neuronowego. Podobno bardziej skuteczny niż algorytm Kohonena, który jest jego szczególnym przypadkiem.
  Neurony są sortowane w zależności od ich odległości od wektora $\mathbf{x}$ a następnie ulegają adaptacji w zależności od ich pozycji $s(i)$ w posortowanym ciągu odległości – wg funkcji wykładniczej $exp(-s(i)/\lambda)$.

Rozdział 9

Mapy SOM — Pakiet SomToolbox2

Pakiet ten działa w środowisku MATLAB 5. Podstawą obliczeń są dwie struktury: data-struct (sD) i som-strukt (sM).

9.1. Tworzenie struktury danych data–struct sD i normalizacja

Strukturę taką można utworzyć na dwa sposoby:

1. sD = som_read_data('iris.dat'); – przez wczytanie danych z pliku Ascii; funkcja może mieć dodatkowy argument (,'x') określający braki w danych czyli *missing values*,
2. sD = som_data_struct(D, 'name', 'iris-sd', 'comp-names', {'SepalL, ... ,'PetalW'}) ; – przez utworzenie struktury sD z MatLabowskiej tablicy danych znajdującej się w przestrzeni roboczej (workspace) MatLaba.

O polach utworzonej w ten sposób struktury sD piszemy dalej w sekcji 9.1.3.

9.1.1. Pierwszy sposób — czytanie danych Ascii

Dane w pliku (np. o nazwie 'iris.data') powinny mieć następującą postać (kropki oznaczają, że opuściliśmy tu pewne fragmenty tych danych):

```
4
#n SepalL SepalW PetalL PetalW
5.1 3.5 1.4 0.2 Setosa
```

```
4.9 3.0 1.4 0.2 Setosa
...
5.0 3.3 1.4 0.2 Setosa
7.0 3.2 4.7 1.4 Versicolor
...
...
5.9 3.0 5.1 1.8 Virginica
```

Strukturę Data-struct zawierającą te dane można utworzyć wykonując instrukcję:
sD=som_read_data('iris.data');

Gdybyśmy nie mieli w pliku nazw osobników (nazw przypisanych poszczególnym wektorom-wierszom tablicy danych), to możemy je dodać później do utworzonej struktury sD za pomocą instrukcji som_label — opisanej w następnej sekcji.

9.1.2. Drugi sposób — korzystanie z tablicy MatLabowskiej D

Jeśli mamy tablicę D znajdującą się w przestrzeni roboczej MatLaba, to możemy strukturę sD utworzyć kolejno za pomocą instrukcji:
sD = som_data_struct(D,'name','iris-sD', 'comp_names', 'SepalL', 'SepalW', 'PetalL','PetalW');

Utworzona w ten sposób struktura danych zawiera nazwy zmiennych ('SepalL, ... , 'PetalW'), ale nie zawiera nazw osobników, tj. wektorów wierszy. Można je dodać za pomocą rozkazu som_label (tutaj każdy wektor-wiersz otrzymuje po prostu nazwę gatunku irysa do którego należy):

sD = som_label(sD,'add',[1:50]','Setosa');
sD = som_label(sD,'add',[51:100]','Versicolor');
sD = som_label(sD,'add',[101:150]','Virginica');

9.1.3. Pola struktury danych sD

Utworzona instrukcjami przedstawionymi w sekcji 9.1.1 lub 9.1.2 struktura sD zawiera następujące pola:

```
sD =
          type: 'som_data'
          data: [150x4 double]
        labels: {150x1 cell}
          name: 'iris.data'
    comp_names: {4x1 cell}
     comp_norm: {4x1 cell}
   label_names: []
```

Pole comp_names zawiera nazwy zmiennych.

Pole comp_norm zawiera informacje o **normalizacji** danych.

9.1.4. Normalizacja i denormalizacja zmiennych

Dopuszcza się następujące możliwości normalizacji:
'var' (na 0-1), tj. odjąć średnią i podzielić przez σ,
'range', tj. na min–max,
'log' ($x^{new} = ln(x - min(x) + 1)$),
'logistic' (softmax; $\hat{x} = (x - \bar{x})/\sigma_x$; $x^{tr} = 1/(1 + exp\{-\hat{x}\})$,
'histD', 'histC' (histogram equalization).

Normalizacja danych wykonuje się za pomocą instrukcji:
sD = som_normalize(sD,'var'); % zamiast 'var' może być inny sposób, np. 'range', 'log', ...

Funkcja som_normalize może zawierać jeszcze trzeci argument określający które zmienne mają być normalizowane. Np.

sD = som_normalize(sD,'log',[1 3]); – zostaną zlogarytmowane tylko pierwsza i trzecia zmienna.

Przykładowa informacja zawarta w 1 komórce pola sD.comp_norm:

```
sM.comp_norm{1}
ans =
      type: 'som_norm'
    method: 'var'
    params: [5.8433 0.8281]
    status: 'done'
```

Normalizacja wykonana na strukturze sD może zostać wykonana w ten sam sposób na innej (nowej) macierzy o nazwie Dn. Należy wydać w tym celu rozkaz:
Dn = som_normalize(Dn, sD);

Denormalizacja danych jest wykonywana za pomocą analogicznych instrukcji:
sD = som_denormalize(sD);
Zostaną unieważnione wszystkie normalizacje wykazane w polu .comp_norm i przywrócone wartości danych sprzed normalizacji.

Zwróćmy uwagę, że dane zostają „zdenormalizowane", ale opis normalizacji pozostaje. Aby usunąć opis normalizacji należy użyć przy denormalizacji kwalifikatora 'remove':
sD = som_denormalize(sD,'remove');

9.2. Tworzenie mapy, instrukcja som_make

9.2.1. Postępowanie standardowe — instrukcja *som_make* z wartościami domyślnymi

Struktura–mapa sM może być utworzona za pomocą rozkazu (zaleca się, żeby dane zostały najpierw znormalizowane):

sM = som_make(sD);

Procedura som_make wywołana bez dalszych parametrów inicjalizuje i trenuje mapę według wartości domyślnych wbudowanych w procedurę.

W ten sposób zostają automatycznie określone rozmiary mapy (na podstawie stosunku wartości własnych m. kowariancji obliczanych danych) a następnie (default) wykonuje uczenie wsadowe (training using batch algorithm) w dwóch fazach (Rough training phase... Fine tuning phase...), po czym zostaje obliczony błąd kwantyzacji i błąd reprezentacji topologicznej (opisane wcześniej w sekcji 8.6).

Te ostatnie mogą zostać obliczone oddzielnie za pomocą rozkazu:

[q,t] = som_quality(sM,D) % for the trained map sM

Dla danych Iris otrzymujemy:

Final quantization error: 0.393 Final topographic error: 0.013

9.2.2. Pola struktury map-struct

Utworzona struktura sM zawiera następujące pola:

```
sM =
          type: 'som_map'
      codebook: [66x4 double]
         topol: [1x1 struct]
        labels: {66x1 cell}
         neigh: 'gaussian'
          mask: [4x1 double]
     trainhist: [1x3 struct]
          name: 'SOM 29-Dec-2000'
    comp_names: {4x1 cell}
     comp_norm: {4x1 cell}
```

Pole codebook zawiera wektory wag; pole labels jest na razie puste (wypełnimy je za chwilę, wykonując som_autolabel), comp_names zawiera nazwy zmiennych, comp_norm – informacje i dane do normalizacji.

Pole codebook zawiera współrzędne wektorów wagowych:

```
sM.codebook
ans =
   -1.4152    0.0229   -1.3285   -1.3285
   -1.3005    0.0068   -1.2852   -1.2746
```

```
    ...       ...       ...       ...
 1.2796    0.4892    1.1476    1.3781
 1.6463    0.5612    1.2880    1.3040
```

Pole topol zawiera informacje o topologii utworzonej mapy:

```
sM.topol
ans =
       type: 'som_topol'
      msize: [11 6]
    lattice: 'hexa'
      shape: 'sheet'
```

Pole labels może zawierać nazwy wektorów wagowych. Na początku, po utworzeniu mapy, pole to jest puste. Odpowiednie nazwy można nadać za pomocą procedury som_autolabel — patrz niżej.

Pole mask zawiera tzw. maskę na zmienne: jeśli elementem maski jest wartość zerowa, to odpowiednia zmienna nie jest uwzględniana w obliczeniach.

```
sM.mask'
ans =     1      1      1      1
```

comp_names to oczywiście nazwy zmiennych:

```
sM.comp_names'
ans =     'SepalL'    'SepalW'    'PetalL'    'PetalW'
```

Pole comp_norm zawiera informacje o normalizacji, takie same jak struktura sD omawiana wcześniej (por. sekcja 9.1.3).

9.2.3. Postępowanie niestandardowe — instrukcja *som_make* z deklarowanymi wartościami parametrów

Ogólna postać wywołania funkcji som_make:

function sMap = som_make(D, varargin);

Dla możliwych argumentów najpierw podaje się identyfikator argumentu, a potem jego wartość. Przykłady:

sMap = som_make(D, [[argID,] value, ...]);
sMap = som_make(D);
sMap = som_make(D, 'munits', 20);
sMap = som_make(D, 'munits', 20, 'hexa', 'sheet');
sMap = som_make(D, 'msize', [4 6 7], 'lattice', 'rect');

Jako D może wystąpić (matrix) training data, size dlen x dim lub (struct) data struct.

Dalszymi argumentami mogą być:

```
'init'       *(string) initialization: 'randinit' or 'lininit' (default)
'algorithm'  *(string) training: 'seq' or 'batch' (default) or 'sompak'
'munits'      (scalar) the preferred number of map units
'msize'       (vector) map grid size
'mapsize'    *(string) do you want a~'small', 'normal' or 'big' map
                       Any explicit settings of munits or msize override this.
'lattice'    *(string) map lattice, 'hexa' or 'rect'
'shape'      *(string) map shape, 'sheet', 'cyl' or 'toroid'
'neigh'      *(string) neighborhood function, 'gaussian', 'cutgauss',
                       'ep' or 'bubble'
'topol'      *(struct) topology struct
'som_topol','sTopol' = 'topol'
'mask'        (vector) BMU search mask, size dim x 1
'name'        (string) map name
'comp_names'  (string array | cellstr) component names, size dim x 1
'tracking'    (scalar) how much to report, default = 1
'training'    (string) 'short', 'default', 'long'
              (vector) size 1 x 2, first length of rough training in epochs,
                       and then length of finetuning in epochs
```

9.3. Procedury som_label i som_autolabel

9.3.1. Procedura som_label

Formalny nagłówek: som_label(sTo, mode, inds, [labels])

Przykłady zastosowań:
sM = som_label(sM, 'add', [1; 10], 'x'); % Dodanie próbkom nr. 1 i 10 nazwy 'x'
sD = som_label(sD, 'clear', 'all'); % usuwa wszystkie nazwy z danych
sD = som_label(sD, 'replace', [1:10]', 'topten');
% zamienia nazwy próbek nr 1–10 na nazwę 'topten'

Ostatnia instrukcja mogłaby być zastąpiona następującymi dwoma:
sD = som_label(sD, 'clear', [1:10]'); sD = som_label(sD, 'add', [1:10]', 'topten');

Jeszcze inne zastosowanie: usunięcie pustych etykiet z wszystkich jednostek mapy:
sD = som_label(sM, 'prune', 'all');

9.3.2. Procedura som_autolabel

Ogólna postać procedury: som_autolabel(sTo, sFrom, [mode], [inds]);
znaczenie: dokąd wstawić, skąd pobrać, sposób, wskaźniki

```
mode: 'add'   -- po prostu dodać, mogą się powtarzać
      'add1'  -- zostaje zapamiętana tylko 1 etykieta
      'freq'  -- dla powtarzających nazw zostaje zapamiętana
                 tylko jedna nazwa i~jej frekwencja
```

```
      'vote'  -- zostaje dodana nazwa o~najw. częstości
                 w~przyp. losowania pierwsza wylosowania
```

Wymienione operacje nie zmieniają starych nazw znajdujących się w sTo.
Przykłady stosowania:
sM = som_autolabel(sM,sD)   % oznakowanie wg danych
sD = som_autolabel(sD,sM)   % oznakowanie wg mapy (codebook vectors)
sD = som_autolabel(sM,sD,'vote',[5])   % etykietuje jednostki mapy
według nazw występujących w 5-tej kolumnie danych, na zasadzie 'vote'.

% moze sie przydac, usuwa z pokazanej mapy wszystkie markery
% (hits, labels trajectories) naniesione przez som_show_add
% i mozna zaczac pokazywanie od nowa
som_show_clear

9.4. Wizualizacja mapy: procedura som_show

9.4.1. Procedury som_show i som_show_add

Standardowe wywołanie: som_show (sM); Wykreśla plansze typu U-mat i mapy w postaci heksagonów — jeżeli nie utworzono mapy o innej topologii. Default: dla wszystkich zmiennych. Można uzyskać wykresy tylko dla niektórych zmiennych, np. dwie mapy u_mat możemy otrzymać za pomocą rozkazu
som_show(sM, 'umat', {[1 2],'1,2 only'},'umat',{[3 4],'3,4 only'});

Przykładowe wykresy są przedstawione na rysunku 9.2 a skrypt, za pomocą którego wyprodukowano te wykresy, jest podany w sekcji 9.4.2.

Można używać zestawów kolorów: (colormap) jet, hsv, hot, gray

Metoda wizualizacji u_mat

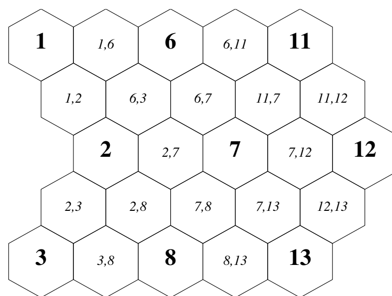
Metoda ta pochodzi od informatyka o nazwisku *Ultsch*. Na mapie widzimy podwójne heksagony. Struktura takiej mapy jest pokazana na rysunku 9.1. Pokazano tam fragment mapy hexagonalnej o wymiarach 5×3.

Mapa *u_mat* pokazuje kolorami:

a) w głównych sześciobokach oznaczonym pojedynczymi liczbami: średnie lub mediany odległości od sąsiadów;

b) w dodatkowych sześciobokach: odległości od wschodniego lub południowego sąsiada.

Na wykresie U-mat wysokie wartości oznaczają granice klasterów; niskie wartości rozłożone obok siebie wskazują na same klastery. Przykładowy wykres średnich odległości między sąsiadami każdego węzła mapy pokazano na rysunku 9.4 jako subplot(2,2,2).



Rysunek 9.1: *zasady wizualizacji* u_mat.

Wykresy typowej mapy składającej się z heksagonów

Ten typ wykresu jest pokazany na rysunku 9.2 jako wykresy górny prawy i obydwa dolne.

Na tego typu wykresach można nanosić różnego rodzaju informacje o wektorach danych które znalazły się w strefie wpływów wektora kodowego stojącego w odpowiedniości z danym heksagonem.

Przykładowo wykres dolny lewy na rysunku 9.2 pokazuje gatunki i frekwencje irysów należących do danego heksagonu, a wykres dolny prawy pokazuje kolorem i wielkością gatunek i frekwencję odpowiednich irysów.

Wykres górny prawy na tym samym rysunku pokazuje tzw. component-plane dla pierwszej cechy irysa (*SepalL*). Tutaj dla każdego heksagonu jest pokazana średnia wartość tej cechy (tj. *SepalL*) irysów skupionych wokół wektora kodowego będącego w odpowiedniości z danym heksagonem.

Wykresy typu component-plane można otrzymać dla każdej lub dla wybranych zmiennych.

Procedura som_show_add

nanosi na wykres — wyprodukowany przez som_show dodatkowe informacje dotyczące etykiet (labels), obsadzenia jednostek (hits) i trajektorii (trajectories).

labels procedura som_autolabel, nadaje etykiety jednostkom, is used to categorize the units (or some units) by giving them names,

hits Hit histograms are actually markers that show the distribution of the best matching units for a given data set (procedura som_hits).

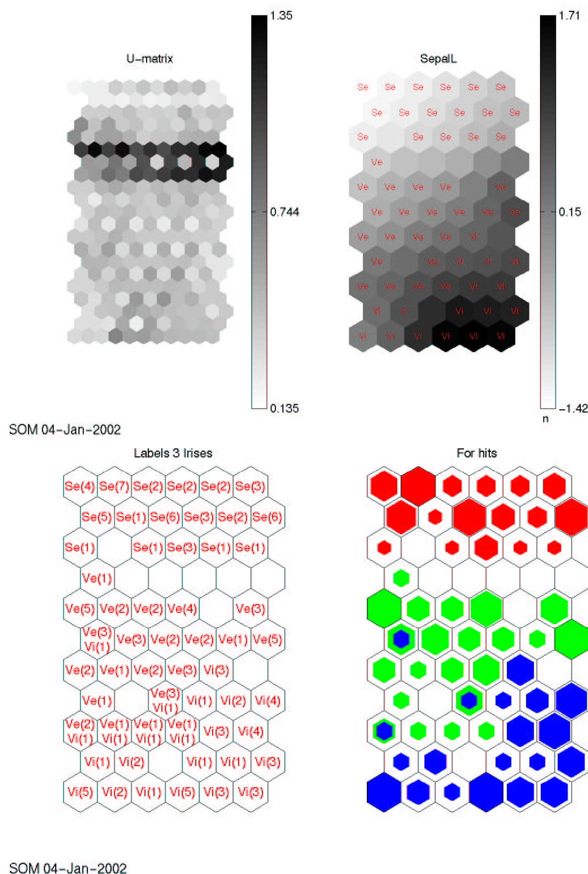
trajectories – show the best matching units for a given data set that is time series (or any ordered series), procedura som_trajectory.

Przykłady użycia
som_show(sM, 'umat', 'all', 'comp', 1:4, 'empty', 'Labels', 'norm','d');
som_show_add('label',sM, 'subplot',6);

9.4.2. Przykładowy skrypt wykorzystujący procedurę som_show

Pokazujemy skrypt który rysuje różne warianty map Kohonena dla danych iris.data znajdujących się w kartotece somtb2

```
 %% job_som1.m
 sD=som_read_data('iris.data'); sD=som_normalize(sD,'var')
     sD=som_label(sD,'replace',[1:50],'Se');
     sD=som_label(sD,'replace',[51:100],'Ve');
     sD=som_label(sD,'replace',[101:150],'Vi');
 sM=som_make(sD)
 sM=som_autolabel(sM,sD,'vote');
 colormap(1-gray); % bedzie w~odcieniach szarosci
 som_show(sM,'umat','all','comp',1);
 som_show_add('label',sM,'TextSize',8,'TextColor', 'r',...
                     'subplot',2);
%% -------------------------------------
 %% job_som2.m
 % Przeczytanie danych, ich normalizacja
 sD=som_read_data('iris.data'); sD=som_normalize(sD,'var')
 % Utworzenie struktury--mapy, nadanie etykiet przez 'freq'
 %    i~pokazanie dwoch map na razie bez etykiet
 sD = som_label(sD, 'replace', [1:50]', 'Se');
 sD = som_label(sD, 'replace', [51:100]', 'Ve');
 sD = som_label(sD, 'replace', [101:150]', 'Vi');
 sM=som_make(sD)
 sM=som_autolabel(sM,sD,'freq');    % 'add', 'add1', 'freq'
 som_show(sM,'empty','Labels 3 Irises', 'empty', 'For hits');
 %    Pokazanie etykiet na mapie
 som_show_add('label',sM,'TextSize',10,'TextColor', 'r','subplot',1);
 %    Obliczenie tzw. trafien ("hits"):
 %      ile probek reprezentuja poszczegolne neurony
 %      liczba trafien zostanie pokazana na mapie przez
 %      wielkosc heksagonow, a~gatunek irisa odmiennym kolorem
 h1=som_hits(sM,sD.data(1:50,:));  h2=som_hits(sM,sD.data(51:100,:));
 h3=som_hits(sM,sD.data(101:150,:));
 som_show_add('hit',[h1, h2, h3],'MarkerColor',...
   [1 0 0; 0 1 0; 0 0 1], 'subplot', 2);
 % Otrzymujemy plansze z~dwoma mapami:  z~etykietami i~trafieniami
%% -------------------------------------
```



Rysunek 9.2: *Dwa przykłady map Kohonena otrzymane za pomocą pakietu* somtoolbox2. Góra: *Mapa u-mat – po lewej, mapa cechy nr 1 (SepalL) po prawej.* Dół: *Nazwy wektorów kodowych po lewej (opcja 'freq), liczby trafień (hits) znakowane wielkością heksagonu – po prawej,*

9.5. Wizualizacja mapy — procedura som_grid

Procedura pełni funkcję podobną do mesh w MatLabie. Przedstawia graficznie dane dwu- i trójwymiarowe zawarte w strukturach sD i sM, jednocześnie pozwala na swobodne operowanie markerami, kolorami i grubościami linii dostępnymi w MatLabie.

9.5.1. Przykładowy skrypt wykorzystujący procedurę som_grid

Pokazujemy użycie procedury som_grid — instrukcje wypisane z modułu demonstracyjnego som_demo2.m

```
 %% -------------------------------------
 %% job_grid.m    -- som_demo2.m
 sD=som_read_data('iris.data');
        % teraz skrocenie etykiet probek
  sD = som_label(sD,'replace',[1:50]','se');
  sD = som_label(sD,'replace',[51:100]','ve');
  sD = som_label(sD,'replace',[101:150]','vi');
  %  wykres typu scatterplot-matrix
  %     Here are the histograms and scatter plots of the four variables.
fh1=figure; k=1;
for i=1:4,
  for j=1:4,
    if i==j,
      subplot(4,4,k); hist(sD.data(:,i)); title(sD.comp_names{i})
    elseif i<j,
      subplot(4,4,k);
      plot(sD.data(:,i),sD.data(:,j),'k.')
      xlabel(sD.comp_names{i})
      ylabel(sD.comp_names{j})
    end
    k=k+1;
  end
end;  drawnow;
      %%  Normalizacja danych i~utworzenie mapy
      %%
 sD = som_normalize(sD,'var');
 sM = som_make(sD);

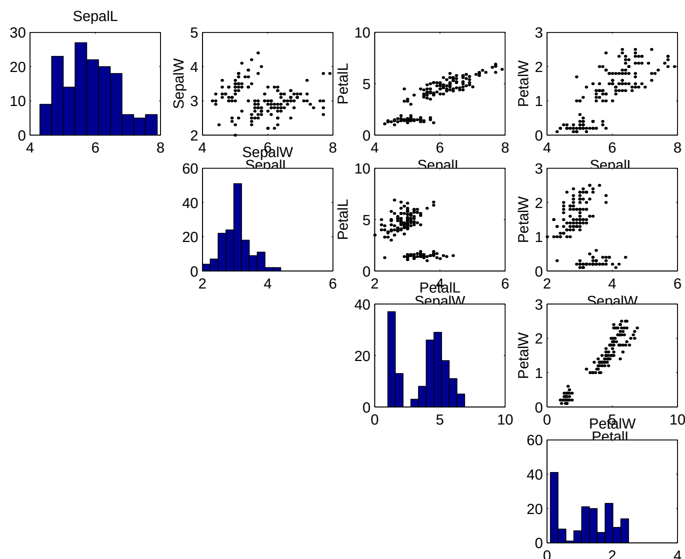
fh2=figure; subplot(2,2,1); %The map grid in the output space.
som_grid(sM,'Linecolor','k');   view(0,-90), title('Map grid')
xlabel('m2'); ylabel('m1');
%     - A surface plot of distance matrix: both color and
%       z-coordinate indicate average distance to neighboring
%       map units. This is closely related to the U-matrix.

subplot(2,2,2)
Co=som_unit_coords(sM); U=som_umat(sM); U=U(1:2:size(U,1),1:2:size(U,2));
som_grid(sM,'Coord',[Co, U(:)],'Surf',U(:),'Marker','none');
view(-80,45), axis tight, title('Distance matrix')
colorbar; xlabel('m2'); zlabel('dist'); ylabel('m1');
%     - The map grid in the output space. Three first components
%       determine the 3D-coordinates of the map unit, and the size
```

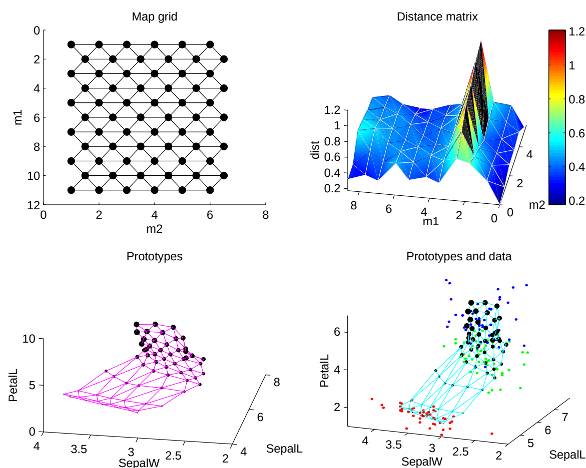
```
%       of the marker is determined by the fourth component.
%       Note that the values have been denormalized.

subplot(2,2,3)
M = som_denormalize(sM.codebook,sM);
som_grid(sM,'Coord',M(:,1:3),'MarkerSize',M(:,4)*2)
view(-80,45),  title('Prototypes')
xlabel('SepalL'); ylabel('SepalW'); zlabel('PetalL');
%     - Map grid as above, but the original data has been plotted
%       also: coordinates show the values of three first components
%       and color indicates the species of each sample.  Fourth
%       component is not shown.

subplot(2,2,4)
som_grid(sM,'Coord',M(:,1:3),'MarkerSize',M(:,4)*2), hold on
D = som_denormalize(sD.data,sD);
plot3(D(1:50,1),D(1:50,2),D(1:50,3),'r.',...
      D(51:100,1),D(51:100,2),D(51:100,3),'g.',...
      D(101:150,1),D(101:150,2),D(101:150,3),'b.')
view(-72,64), axis tight, title('Prototypes and data')
xlabel('SepalL'); ylabel('SepalW'); zlabel('PetalL');
%%
%%---------------------------------------------------
```



Rysunek 9.3: *Wykres typu scatterplot-matrix dla 4 cech irysa, otrzymany za pomocą skryptu job_grid.m jako figure fh1*



Rysunek 9.4: *Fragmenty rysunków wykonywanych przez moduł* som-demo2 *pakietu* somtoolbox*. Siatka mapy, odległości między węzłami mapy, oraz wykresy trójwymiarowe wektorów kodowych oraz danych. Wykonane za pomocą skryptu* som_grid

Rozdział 10

Wizualizacja wielowymiarowych danych: PCA i Odwzorowanie Sammona

Przedstawiamy tu dwie metody wizualizacji punktów znajdujących się w wielowymiarowej przestrzeni. Są to: metoda składowych głównych i metoda odwzorowań Sammona. Obydwie metody pozwalają dokonać specyficznej redukcji wymiarowości do przestrzeni o niższej liczbie wymiarów, w szczególności do $d = 2$ lub $d = 3$ wymiarów. Przetransformowane dane mogą być wtedy zobrazowane na płaszczyźnie lub w przestrzeni 3-wymiarowej przy użyciu zwykłych sposobów.

Metoda PCA wykonuje transformacje liniowe, a metoda Sammona — transformacje nieliniowe danych.

Obydwie metody są zrealizowane w pakiecie somtoolbox2.

10.1. Metoda PCA czyli składowych głównych

10.1.1. Ogólne zasady wyznaczania składowych głównych

Metoda składowych głównych jest jedną z bardziej rozpowszechnionych metod analizy danych wielowymiarowych, szczególnie w kontekście redukcji wymiarowości i wizualizacji. Dobry i zrozumiały opis tej metody znajduje się w książkach Morrisona [6] oraz Johnsona i Wicherna [17]; istnieje również świetna monografia Iana Jolliffe'a [4].

W środowisku statystycznym metoda składowych głównych jest wiązana z nazwiskiem Hotellinga; w środowisku technicznym (przetwarzanie sygnałów, sieci neuronowe) metoda ta jest określana mianem transformaty Karhunen–Loevego.

Przypuśćmy, że mamy daną tablicę $\mathbf{X}$ o wymiarach $[n \times d]$. Wektory-wiersze tej tablicy mogą być interpretowane jako punkty $\in R^d$. Przyjmijmy — dla wprowadzającej ilustracji — że $d = 2$. Wtedy wektory-wiersze tablicy $\mathbf{X}$ można zobrazować na płaszczyźnie $<x1, x2>$. Nanosząc dane n punktów (określonych w tablicy $\mathbf{X}$) na tę płaszczyznę otrzymujemy tzw. *chmurę punktów indywidualnych*. Na ogół, dla danych statystycznych spotykanych w praktyce, tzn. **nie** wygenerowanych za pomocą specjalnych modeli, chmura punktów indywidualnych przypomina elipsę (w przypadku $d = 3$ elipsoidę, a w przypadku $d > 3$ hiper-elipsoidę).

Kierunki osi głównych tej elipsy lub (hiper) elipsoidy wyznacza się z równania macierzowego

$$(\mathbf{S}_X - \lambda\mathbf{I})\mathbf{a} = \mathbf{0}, \tag{10.1}$$

gdzie $\mathbf{a} = [a_1, \ldots, a_d]^T$ jest jednym z kierunków głównych, a $\mathbf{S}_X$ jest macierzą kowariancji danych zapisanych w tablicy $\mathbf{X}$.

Dla jednoznaczności rozwiązania równania (10.1) przyjmiemy, że wektor $\mathbf{a}$ jest wektorem o długości 1, co oznacza że $\mathbf{a}^T\mathbf{a} = 1$.

Wektory $\mathbf{a}$ spełniające równanie (10.1) i wyznaczające kierunki główne są poszukiwane sekwencyjnie.

Najpierw szuka się wektora $\mathbf{a}_1$ wyznaczającego najdłuższą oś elipsy lub (hiper) elipsoidy. Wektor $\mathbf{a}_1$ musi spełniać równanie macierzowe (10.1). Okazuje się, że równanie to ma d rozwiązań, odpowiadających d wartościom własnym macierzy $\mathbf{S}$. Tak więc, rozwiązując równanie (10.1) otrzymujemy d wartości własnych. Porządkując otrzymane wartości własne w ciąg nierosnący otrzymujemy

$$\lambda_1 \geqslant \lambda_2 \geqslant \ldots \geqslant \lambda_d \geqslant 0.$$

Z każdą z tych wartości własnych jest związany odpowiadający jej wektor własny

$$\mathbf{a}_1,\ \mathbf{a}_2,\ \ldots\ ,\ \mathbf{a}_d.$$

Można wykazać, że kierunek najdłuższej osi elipsoidy jest identyczny z kierunkiem wyznaczanym przez wektor $\mathbf{a}_1$. Dalsze osie elipsoidy są wyznaczane przez dalsze wektory własne.

Wszystkie wektory własne możemy zestawić w macierz $\mathbf{A}$:

$$\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a_d}].$$

Można wykazać, że macierz $\mathbf{A}$ wyznacza transformację będącą rotacją układu współrzędnych $<x1, \ldots, xd>$ (zaczepionego w punkcie $(\overline{x1}, \ldots, \overline{xd})$) do układu wektorów $<\mathbf{e}_1, \ldots, \mathbf{e}_d>$ wyznaczających osie główne elipsoidy.

Rotacja ta może być użyta do transformacji macierzy $\tilde{\mathbf{X}}$. Otrzymamy wtedy:

$$\mathbf{Y} = \tilde{\mathbf{X}}\mathbf{A}. \tag{10.2}$$

Otrzymana w ten sposób tablica **Y** przedstawia te same dane co tablica **X**, ale wyrażone w innym układzie współrzędnych.

Definicja. Elementy tablicy **Y** określonej wzorem (10.2) są nazywane współrzędnymi głównymi (ang. *principal coordinates*) tablicy danych **X**.

Macierz kowariancyjna „nowych” danych **Y** równa się

$$\mathbf{S}_Y = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & \lambda_p \end{bmatrix}. \tag{10.3}$$

Ze wzoru (10.3) wynika, że przetransformowane dane mają następujące własności:

- wariancje wartości zapamiętanych w kolejnych kolumnach wynoszą odpowiednio $\lambda_1, \lambda_2, \ldots, \lambda_p$,
- pary kolumn tablicy **Y** przedstawiają wartości zmiennych nieskorelowanych.

Można pokazać, że

$$trace(\mathbf{S}_X) = \sum_{i=1}^{d} \underbrace{s_{ii}}_{elementy\ przek.} = \sum_{i=1}^{d} \underbrace{s_i^2}_{wariancje\ x1,\ldots,xd} = \sum_{i=1}^{d} \lambda_i.$$

Wynika stąd, że *suma wariancji współrzędnych głównych równa się sumie wariancji zmiennych wyjściowych danych w układzie* $< x1, \ldots, xp >$.

Jeżeli suma dwóch lub trzech pierwszych wartości własnych jest duża (np. stanowi 90% lub więcej całkowitej ich sumy), to znaczy że wariancje wyjściowych zmiennych $X_1, \ldots, X_p$ są odtwarzane w znacznym stopniu przez nowe (przetransformowane) zmienne, czyli przez współrzędne główne. Można stąd dedukować, że elipsoida w R^d obejmująca dane **X** jest faktycznie mocno spłaszczona, wobec czego jest możliwe i uzasadnione posługiwanie się reprezentacją tych danych w niższej przestrzeni R^r wyznaczonej przez r pierwszych składowych głównych.

Zauważmy również, że każda współrzędna główna (składowa główna) jest otrzymywana jako kombinacja liniowa *wszystkich* zmiennych wyjściowych:

$$\mathbf{y}^{(j)} = \tilde{\mathbf{X}} \times \mathbf{a}_j. \tag{10.4}$$

10.1.2. Wskazówki praktyczne i podsumowanie

Przy obliczaniu współrzędnych głównych (składowych głównych) należy obejrzeć koniecznie wartości własne odpowiadające tym składowym.

Tylko wtedy, gdy udział pierwszych wartości własnych w stosunku do sumy wszystkich wartości własnych jest duży, mamy teoretyczne podstawy (i uzasadnienie, jesteśmy uprawnieni) do zastąpienia tablicy danych $\mathbf{X}$ o d kolumnach zredukowaną macierzą $\mathbf{Y}^{(r)}$ o r kolumnach ($r < d$).

Podsumowanie

Współrzędne główne (składowe główne) są to współrzędne w układzie osi głównych elipsy lub (hiper) elipsoidy koncentracji przybliżającej chmurę punktów indywidualnych w R^d.
Współrzędne te są dane wzorem

$$\mathbf{Y} = \tilde{\mathbf{X}}\mathbf{A}\ ,$$

gdzie $\mathbf{A}$ zawiera kolumnami wektory własne macierzy kowariancji $\mathbf{S}_X$, a $\tilde{\mathbf{X}}$ jest scentrowaną tablicą danych.

Jeżeli — wskutek współzależności analizowanych cech — elipsoida koncentracji ma kształt mocno spłaszczony, wtedy jest uzasadnione zastąpienie d współrzędnych oryginalnych (zapisanych w tablicy $\mathbf{X}$) mniejszą liczbą r współrzędnych wziętych jako pierwsze r kolumny z tablicy $\mathbf{Y} = \tilde{\mathbf{X}}\mathbf{A}$.

Utrata informacji jest mierzona ilorazem

$$\sum_{j=r+1}^{d} \lambda_j / \sum_{j=1}^{d} \lambda_j\ ,$$

przy czym

$$\lambda_1 + \cdots \lambda_d = s_1^2 + \cdots + s_d^2,$$

gdzie s_j^2 ($j = 1, \ldots, d$) oznacza wariancję zmiennej X_j.

10.1.3. Obliczenia PCA za pomocą pakietu *somtoolbox*

Obliczenia składowych głównych odbywają się za pomocą funkcji pcaproj. Przykładowe wywołania tej funkcji:

(•) [P,V,me,l] = pcaproj(D, odim); – gdy liczymy po raz pierwszy,

lub

(•) P = pcaproj(D, V, me); – gdy znamy już operatory rzutowania.

Jako argument funkcji pcaproj należy podać przede wszystkim tablicę danych którą chcemy zrzutować do układu współrzędnych głównych. Może to być tablica D według standardu MatLaba; można tu również posłużyć się odpowiednimi

strukturami data_struct **SD** lub map_struct **sM** — specyficznymi dla pakietu somtoolbox.

(•) Pierwsze, standardowe wywołanie, dostarcza dla zadeklarowanej tablicy D *odim* współrzędnych głównych. Obliczenia procedury są w tym przypadku następujące:

Najpierw — dla zadeklarowanej tablicy **D** — oblicza się wektor średnich **me** oraz macierz kowariancji $\mathbf{S}_X$.

Następnie wyznacza się wszystkie wartości własne i wektory własne macierzy $\mathbf{S}_X$. Obliczone wartości własne — wraz z odpowiadającymi im wektorami własnymi — są porządkowane malejąco.

Wreszcie oblicza się — według wzoru (10.4) — żądanych *odim* współrzędnych głównych, które umieszcza się jako wynik w tablicy **P** o wymiarach $[n \times odim]$.

Oprócz tego w wynikach mogą być umieszczone:

- ⋄ **V** o wymiarach $[d \times odim]$ – wektory własne użyte do projekcji (pierwszych *odim* kolumn m. **A** używanej w naszym opisie metody),
- ⋄ **me** o wymiarach $[1 \times d]$ – wektor średnich,
- ⋄ **l** o wymiarach $[1 \times odim]$ – frakcje $|\lambda_j|/\sum_j |\lambda_j|$.

(•) Obliczone przy pierwszym wywołaniu wektory własne mogą służyć, wraz z wektorem średnich, do projekcji innego zbioru danych, np. zapamiętanego w tablicy **D1** – oczywiście o tej samej liczbie zmiennych, co tablica **D** z której wyznaczono wektory **V**.

Rzuty dla innego zbioru danych, np. **D1** możemy otrzymać za pomocą rozkazu:

```
P1 = pcaproj(D1, V, me)
```

10.1.4. Przykładowy skrypt do obliczeń współrzędnych głównych za pomocą pakietu *somtoolbox*

Podany niżej skrypt oblicza najpierw 3 pierwsze składowe główne dla znormalizowanych danych zapamiętanych w strukturze **sD**. Ponieważ dane są znormalizowane, obliczona macierz kowariancji jest faktycznie macierzą korelacji rozważanych zmiennych. Frakcje odtworzonej sumy $\sum_{i=1}^{4} \lambda_i$ przez kolejne składowe wynoszą odpowiednio: 0.7272, 0.2303, 0,0368, a skumulowane sumy tych frakcji: 0.7272, 0.9575, 0.9943

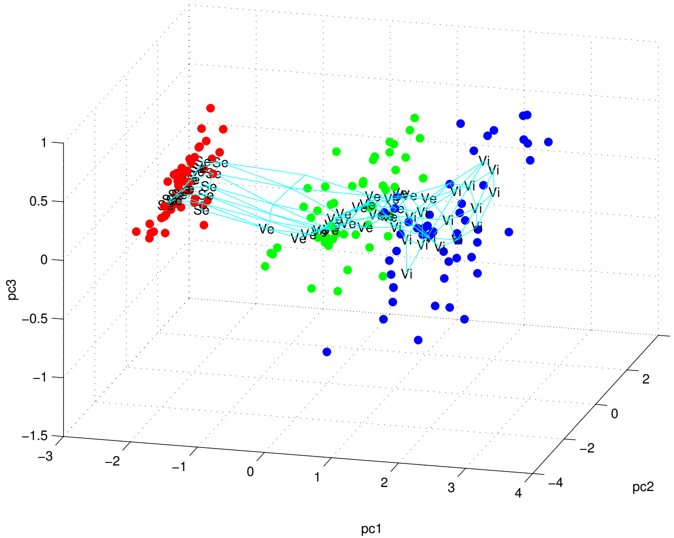
Najpierw rzutujemy do wyznaczonego układu kierunków głównych wektory kodowe zapamiętane w strukturze irysowej sM. Rzutowane wektory kodowe są *znormalizowane.*

Wykres współrzędnych głównych jest robiony za pomocą procedury som_grid — taki wykres pokazuje powiązania wektorów kodowych odpowiadające sąsiedztwu na mapie sM.

Na ten sam trójwymiarowy wykres zostają zrzutowane — za pomocą obliczonego już operatora rzutowania czyli tablicy **V** wektory kodowe zapamiętane w strukturze sD. Jest to 150 punktów $\in R^d$ odpowiadających poszczególnym kwiatom irysa. Gatunki poszczególnych kwiatów są zaznaczone różnymi kolorami: czerwony oznacza irysy Setosa (Se), zielony – Versicolor (Ve), a niebieski – Virginica (Vi). *Som_quality: q=0.3927, t=0.0133.*

Odpowiedni wykres jest pokazany jako Rysunek 10.1.

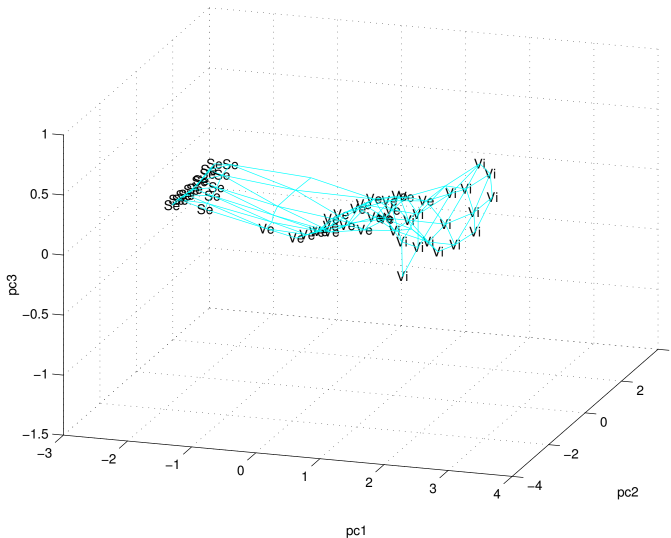
Wykres samych wektorów kodowych (znormalizowanych) jest pokazany na rysunku 10.2. Rysunek ten jest sporządzony w tym samym układzie współrzędnych, co rysunek 10.1, tj. w układzie składowych głównych $pca1, pca2, pca3$ wyznaczonych ze znormalizowanych danych obejmujących 150 kwiatów irysa zapamiętanych w strukturze **sD**. *Som_quality: q=0.3927, t=0.0133.*



Rysunek 10.1: *Wykres danych ze struktury* sD *(punkty zaznaczone trzema kolorami w zależności od gatunku irysa) oraz wektorów kodowych ze struktury* sM *(punkty połączone siatką) w układzie składowych głównych pca1,pca2,pca3. Układ współrzędnych obliczony ze znormalizowanych danych irysa*

A oto skrypt, za pomocą którego wykonano oba rysunki:

```
%%------------------ job_pca.m ----------------------------------
% make the data  and the SOM
sD = som_read_data('iris.data'); sD = som_normalize(sD,'var');
sD = som_label(sD, 'replace', [1:50], 'Se');
```



Rysunek 10.2: *To samo, co na rysunku 10.1, ale pokazano tylko siatkę wektorów kodowych wraz z ich etykietami*

```
sD = som_label(sD, 'replace', [51:100], 'Ve');
sD = som_label(sD, 'replace', [101:150], 'Vi');
sM = som_make(sD); sM = som_autolabel(sM,sD,'vote');
% pause
% find PCA-projection of the data -- from correlation matrix
[Pd,V,me,lambda] = pcaproj(sD,3);
      % coordinates, eigen-vectors, center, lambda
% plot the map grid projection
         % from help:  [P,V,me,l] = pcaproj(D, odim)
         %             or      P     = pcaproj(D, V, me)
fh1=figure
som_grid(sM,'Coord',pcaproj(sM,V,me),'marker','none', ...
 'LineColor','c','Label',sM.labels,'LabelSize',10,'labelcolor','k');
xlabel('pc1'), ylabel('pc2'), zlabel('pc3')
% plot also the original data with color indicating subspecies
hold on, grid on
colD = [repmat([1 0 0],50,1); repmat([0 1 0],50,1);...
       repmat([0 0 1],50,1)];
som_grid('rect',[150 1], 'Line', 'none','Coord',Pd,'markercolor',colD)
%%
% make another map containing projections of codebook vectors only
% but using the projections obtained from the normalized data set
```

```
fh2=figure
som_grid(sM,'Coord',pcaproj(sM,V,me),'marker','none', ...
 'LineColor','c','Label',sM.labels,'LabelSize',10,'labelcolor','k');
xlabel('pc1'), ylabel('pc2'), zlabel('pc3')
axis([-3, 4, -4, 4, -1.5, 1]); grid on
%
%% ------------------------------------------------------------
```

10.2. Odwzorowanie Sammona

10.2.1. Omówienie algorytmu odwzorowania Sammona

Mamy n wektorów d-wymiarowych $\mathbf{x}_i$ $(i = 1, 2, ..., n)$ które można interpretować jako punkty osobnicze leżące w przestrzeni R^d. Chcemy zrzutować te punkty do przestrzeni M-wymiarowej $(M = 2, 3)$; odpowiednie rzuty będziemy oznaczać jako $\mathbf{y}_i = [y_1, \ldots, y_M]^T$. Tak więc dla każdego punktu $\mathbf{x}_i$ należy znaleźć jego rzut $\mathbf{y}_i$:

$$\mathbf{x}_i \Longrightarrow \mathbf{y}_i\,, \quad \text{gdzie } \ \mathbf{x}_i \in R^d,\ \mathbf{y}_i \in R^M.$$

W obu przestrzeniach (tj. R^d i R^M) określamy pojęcie odległości $d(i, j) = d_{ij}$ między punktami o numerze i oraz j leżącymi w tej przestrzeni. Do określenia odległości można zastosować dowolną metrykę, w szczególności euklidesową. W przypadku przyjęcia tej ostatniej odległość między dwoma punktami definiuje się jako pierwiastek kwadratowy z sumy kwadratów różnic liczonych po wszystkich składowych tych punktów.

Wprowadźmy następujące oznaczenia:

$D^*_{ij} = D(\mathbf{x}_i, \mathbf{x}_j)$ – odległości między punktami w przestrzeni R^d,

$D_{ij} = D(\mathbf{y}_i, \mathbf{y}_j)$ – odległości między odpowiednimi rzutami w przestrzeni R^M.

Zadanie odwzorowania nieliniowego Sammona polega na takim doborze wektorów $\mathbf{y}_i$, aby zminimalizować funkcję błędu E zdefiniowaną w postaci

$$E = \frac{1}{c} \sum_{i<j}^{n} \left| D^*_{ij} - D_{ij} \right|^2 / D^*_{ij}, \tag{10.5}$$

przy czym

$$c = \sum_{i<j}^{n} D^*_{ij},$$

$$D_{ij} = \sqrt{\sum_{s=1}^{M} (y_{is} - y_{js})^2}, \quad D^*_{ij} = \sqrt{\sum_{s=1}^{d} (x_{is} - x_{js})^2}\ .$$

W minimalizacji funkcji błędu E określonej wzorem (10.5) Sammon zastosował iteracyjną metodę optymalizacyjną Newtona, uproszczoną do postaci

$$y_{pq}(k+1) = y_{pq}(k) - \eta\Delta_{pq}(k), \tag{10.6}$$

w której

$$\Delta_{pq}(k) = \frac{\frac{\delta E}{\delta y_{pq}}}{\left|\frac{\delta^2 E}{\delta^2 y_{pq}^2}\right|},$$

k oznacza numer iteracji,
$\Delta_{pq}(k)$ jest ilorazem odpowiedniej składowej gradientu i diagonalnego składnika hesjanu — wyznaczonych w k-tej iteracji,
η jest współczynnikiem uczenia; najczęściej nadaje się mu wartości z przedziału $[0.3; 0.4]$.

Przy definicji funkcji błędu w postaci (10.5) odpowiednie składowe gradientu i hesjanu są dane wzorami: (źródło: Osowski, str. 267–268)

$$\frac{\delta E}{\delta y_{pq}} = -\frac{2}{c}\sum_{j=1, j\neq p}^{n}\left[\frac{D^*_{pj} - D_{pj}}{D_{pj}D^*_{pj}}\right][y_{pq} - y_{jq}], \tag{10.7}$$

$$\frac{\delta^2 E}{\delta^2 y_{pq}^2} = -\frac{2}{c}\sum_{j=1, j\neq p}^{n}\frac{1}{D_{pj}D^*_{pj}}\left[(D^*_{pj} - D_{pj}) - \frac{(y_{pq} - y_{jq})^2}{D_{pj}}\left(1 + \frac{D^*_{pj} - D_{pj}}{D_{pj}}\right)\right]. \tag{10.8}$$

Odwzorowanie Sammona jest odwzorowaniem nieliniowym punktów z przestrzeni R^d na odpowiednie ich „rzuty” leżące w przestrzeni R^M.

10.2.2. Obliczenia odwzorowania Sammona za pomocą pakietu *somtoolbox* i przykładowy skrypt

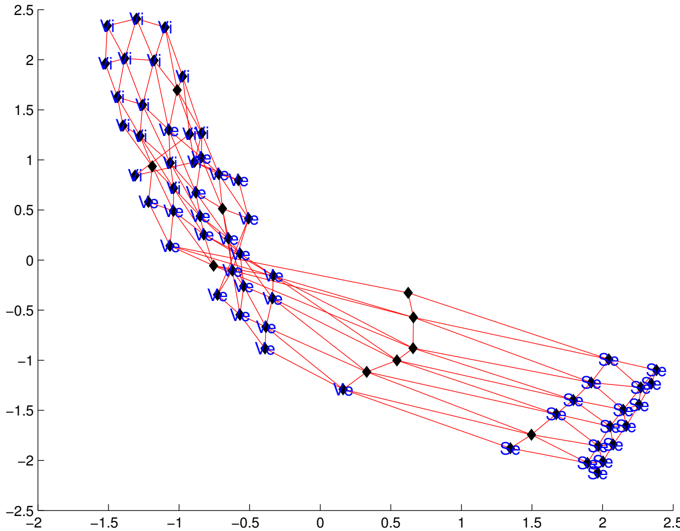
Odwzorowaniu mogą podlegać zarówno wektory danych jak i wektory wagowe reprezentujące klastery danych.

Procedura sammon zaimplementowana w pakiecie somtoolbox2 może być stosowana zarówno do tablicy danych **D**, struktury danych czyli data_struct **sD** jak również do wektorów kodowych znajdujących się w strukturze som_struct **sM**.

Przykładowe wywołania procedury:
P = sammon(D,2); P = sammon(sM,3); P = sammon(sM,3,[],[],[],Md);

Po obliczeniu tablicy projekcji P może nastąpić wizualizacja graficzna przetransformowanych danych znajdujących się w tablicy P za pomocą procedury som_grid lub też innych procedur wizualizacyjnych MatLaba.

Przykładowe wywołanie procedury som_grid:
som_grid(sM,'Coord',P); lub też, z bardziej wyspecyfikowanymi parametrami:



Rysunek 10.3: *Znormalizowane Wektory kodowe danych 'Iris' zrzutowane metodą Sammona na dwu-wymiarową płaszczyznę* $< y1, y2 >$ *(M = 2)*

som_grid(sM,'Coord',sammon(sM,2),'LineColor','r','Label',sM.labels, ...
'labelColor','b', 'Marker','d', MarkerColor, 'k')

Otrzymamy wtedy rysunek 10.3.

Na rysunku tym obserwujemy zjawisko *skręcania się* siatki (mapy). Obraz taki jest dość częstym zjawiskiem przy stosowaniu odwzorowań Sammona. Wynika on z trudności otrzymania obrazu dwuwymiarowego który ma pokazać odległości między punktami w przestrzeni wielowymiarowej, a więc w przestrzeni o nieporównywalnie bogatszych możliwościach.

Czasami pomaga wystartowanie z innej mapy (tj. uzyskanej z innego przybliżenia początkowego przy uczeniu mapy, lub innego układu neuronów na mapie).

A oto przykładowy skrypt:

```
%% -------------------- job_samm.m --------------------------------
% make the data, shorten labels,  make the SOM
sD = som_read_data('iris.data'); sD = som_normalize(sD,'var');
sD = som_label(sD, 'replace', [1:50], 'Se');
sD = som_label(sD, 'replace', [51:100], 'Ve');
sD = som_label(sD, 'replace', [101:150], 'Vi');
sM = som_make(sD); sM = som_autolabel(sM,sD,'vote');
%sM = som_make(sD); sM = som_autolabel(sM,sD,'vote');
% visualization  of codebook vectors by Sammon projection
```

```
% Ps=sammon(sM,3);
som_grid(sM,'Coord',sammon(sM,3),'marker','none',...
         'Label',sM.labels,'labelcolor','k');
%-----------------------------------------------------------------
```

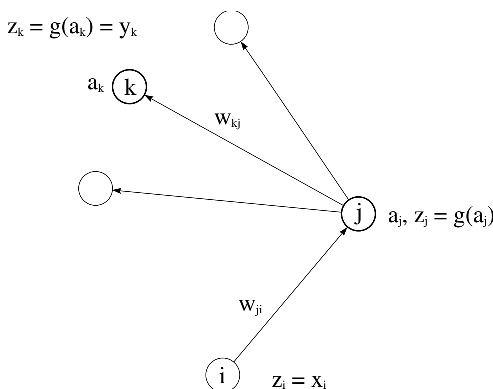
Rozdział 11

Backpropagation — propagacja wsteczna

11.1. Oznaczenia wstępne

Rozpatrzmy sieć dwuwarstwową, składającą się z warstwy wejściowej (warstwa „0"), warstwy pośredniej (ukrytej) „1" i warstwy wyjściowej „2".

Niech symbole i, j, k oznaczają numery jednostek (neuronów) należących do tych warstw, jak to pokazano na rysunku 11.1.



Rysunek 11.1: *Przekazywanie sygnałów od neuronu i na wejściu do neuronu j warstwy pośredniej, a stamtąd do neuronu k na wyjście*

Przy określaniu par $< i, j >$ oraz $< j, k >$ obowiązują następujące zasady:

- Porządek występowania elementów w parach jest ściśle określony.
- Pierwszy element należy do warstwy niższej (wcześniejszej), drugi do warstwy następnej, wyższej.
- Połączenie między parami jest skierowane; bodźce (sygnały) są przekazywane od jednostki stojącej na początku pary (tj. jednostki z niższej warstwy) do jednostki stojącej na końcu pary (tj. jednostki umiejscowionej w wyższej warstwie).

Wymienione fakty można prześledzić na rysunku 11.1.

11.2. Przekazywanie bodźców

Popatrzmy teraz na elementy i, j, k pod kątem ich aktywacji i przekazywanych dalej bodźców.

Neuron i należący do warstwy „0" ma aktywację $z_i^{(0)} = x_i$.

Neuron j zbiera bodźce od I jednostek z warstwy poprzedniej i sumuje je z odpowiednimi wagami. Oznacza to obliczanie sumy

$$a_j^{(1)} = \sum_{i \in \mathcal{I}} w_{ji}^{(1)} z_i^{(0)},$$

gdzie $w_{ji}^{(1)}$ oznacza wagę połączenia skierowanego od jednostki i do jednostki j; wskaźnik $^{(1)}$ u góry oznacza, że są to wielkości obliczone przez neuron umiejscowiony w warstwie „1".

Na podstawie tak wyznaczonej sumy $a_j^{(1)}$ zostaje wyznaczona aktywacja j-tego neuronu

$$z_j^{(1)} = g(a_j^{(1)}),$$

gdzie $g(.)$ oznacza funkcję aktywacji przyjętą dla warstwy „1".
Obliczona aktywacja $z_j^{(1)}$ stanowi bodziec (sygnał) przekazywany dalej (forwards, czyli w przód) do następnej warstwy, w szczególności do k-tej jednostki tej warstwy (patrz rysunek 11.1).

Neuron k zbiera bodźce od J jednostek pochodzących z warstwy poprzedniej (na rysunku 11.1 ma ona numer „1") i sumuje je z odpowiednimi wagami. Oznacza to obliczenie sumy

$$a_k^{(2)} = \sum_{k \in \mathcal{K}} w_{kj}^{(2)} z_j^{(1)},$$

gdzie $w_{kj}^{(2)}$ oznacza wagę połączenia w parze $< j, k >$, a wskaźnik $^{(2)}$ u góry oznacza, że są to wielkości obliczone przez neuron umiejscowiony w warstwie „2".

Na podstawie tak wyznaczonej sumy $a_k^{(2)}$ zostaje wyznaczona aktywacja k-tego neuronu

$$z_k^{(2)} = \tilde{g}(a_k^{(2)}),$$

gdzie $\tilde{g}(a_k^{(2)})$ oznacza funkcję aktywacji przyjętą dla warstwy „2". Może to być ta sama funkcja, którą przyjęto dla warstwy „1", w takim przypadku $\tilde{g}(\cdot) \equiv g(\cdot)$.

Jeżeli neuron nr k jest neuronem warstwy wyjściowej, to obliczona aktywacja $z_k^{(2)}$ jest wynikiem końcowym y_k i jest przekazywana na wyjście jako wynik y_k.

Rezultaty tych działań są podsumowne w tabelce poniżej:

Warstwa	Neurony	Formuły obliczeniowe		
warstwa „2"	neuron ○ k	$a_k^{(2)} = \sum_k w_{kj} z_j^{(1)}$,	$z_k^{(2)} = \tilde{g}(a_k^{(2)})$,	$y_k = z_k^{(2)}$
	↖↑			
warstwa „1"	neuron ○ j	$a_j^{(1)} = \sum_j w_{ji} z_i^{(0)}$,	$z_j^{(1)} = g(a_j^{(1)})$	
	↖↑			
warstwa „0"	neuron ○ i	$z_i^{(0)} = x_i$		

Tabela 11.1:

11.3. Określenie wektora wag

W omawianym wyżej modelu wagi występują w 2 warstwach: W warstwie pierwszej mamy wagi wynikające z połączeń I jednostek warstwy poprzedniej z j-tą jednostką $(j = 1, \ldots, J)$ obecnie analizowanej warstwy pierwszej

$$\mathbf{w}_1^{(1)}, \ldots, \mathbf{w}_J^{(1)}, \quad \text{gdzie} \quad \mathbf{w}_j^{(1)} = [w_{j1}^{(1)}, \ldots, w_{jI}^{(1)}]^T.$$

Analogicznie, dla warstwy drugiej mamy wagi

$$\mathbf{w}_1^{(2)}, \ldots, \mathbf{w}_K^{(2)}, \quad \text{gdzie} \quad \mathbf{w}_k^{(2)} = [w_{k1}^{(2)}, \ldots, w_{kJ}^{(2)}]^T.$$

Wszystkie te wektory wag możemy zestawić w jeden wspólny wektor $\mathbf{w}$ określony następująco:

$$\mathbf{w} = [(\mathbf{w}_1^{(1)})^T, \ldots, (\mathbf{w}_J^{(1)})^T, (\mathbf{w}_1^{(2)})^T, \ldots, (\mathbf{w}_K^{(2)})^T]^T.$$

11.4. Funkcja błędu

Funkcja błędu E wyznaczona dla N próbek jest zdefiniowana jako suma błędów zaobserwowanych dla każdego wzorca n, $(n = 1, \ldots, N)$:

$$E = \sum_n E^n,$$

gdzie E^n jest błędem dla wzorca n.

Błąd E^n może być definiowany na rozmaite sposoby. Jedną z częściej stosowanych definicji błędu jest definicja oparta na zasadzie najmniejszych kwadratów, tj.

$$E^n = \frac{1}{2}\sum_{k=1}^{K}(y_k^n - t_k^n)^2,$$

gdzie $y_k^n = y_k^n(\mathbf{x}^n, \mathbf{w})$ jest wynikiem wyprodukowanym przez sieć (sygnałem postsynaptycznym) na wyjściu k jako odpowiedź na dany sygnał $\mathbf{x}^n$ (sygnał presynaptyczny) – przy użyciu wektora wag $\mathbf{w}$ charakteryzujących siłę połączeń między neuronami.

Błąd E^n dla n-tego wzorca możemy uszczegółowić, wyrażając go za pomocą parametrów sieci przedstawionych na rysunku 11.1:

$$\begin{aligned} E^n &= \frac{1}{2}\sum_{k=1}^{K}(\tilde{g}(\underbrace{a_k^{(2)}}_{\sum_j w_{kj}^{(2)} z_j^{(1)}}) - t_k^n)^2 = \frac{1}{2}\sum_{k=1}^{K}(\tilde{g}(\sum_j w_{kj}^{(2)} \underbrace{z_j^{(1)}}_{g(a_j^{(1)})}) - t_k^n)^2 \\ &= \frac{1}{2}\sum_{k=1}^{K}(\tilde{g}(\sum_j w_{kj}^{(2)} g(\underbrace{a_j^{(1)}}_{\sum_i w_{ji}^{(1)} z_i^{(0)}})) - t_k^n)^2. \end{aligned}$$

Jak wiadomo, proces uczenia przebiega w ten sposób, że na podstawie prezentowanych wzorców sieć adaptuje (koryguje) swoje wagi tak, aby uczynić ostateczny błąd E możliwie najmniejszym.

W jaki sposób sieć ma to zrobić, tzn. jak skorygować swoje wagi aby osiągnąć minimum funkcji błędu?

Na ogół, jeśli funkcja jest różniczkowalna, robi się to metodami gradientowymi. Metody te wymagają obliczenia pochodnych minimizowanej funkcji.

11.5. Obliczanie pochodnych funkcji błędu względem wag w_{ji} oraz w_{kj}

Wprowadźmy oznaczenia

$$\delta_j^{(1)} = \frac{\partial E^n}{\partial a_j^{(1)}}, \quad \delta_k^{(2)} = \frac{\partial E^n}{\partial a_k^{(2)}}.$$

Obliczanie pochodnych względem wag drugiej warstwy

$$\frac{\partial E^n}{\partial w_{kj}^{(2)}} = \underbrace{\frac{\partial E^n}{\partial a_k^{(2)}}}_{\delta_k^{(2)}} \cdot \frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}} = \delta_k^{(2)} \cdot z_j^{(1)}, \tag{11.1}$$

gdzie

$$\delta_k^{(2)} = \left.\frac{\partial E^n}{\partial a_k^{(2)}}\right|_{a_k^{(2)}} = \frac{\partial E^n}{\partial y_k} \cdot \frac{\partial y_k}{\partial a_k^{(2)}} = \frac{\partial E^n}{\partial y_k} \cdot \frac{\partial \tilde{g}(a_k^{(2)})}{\partial a_k^{(2)}}, \tag{11.2}$$

ponieważ $y_k = \tilde{g}(a_k^{(2)})$.

Obliczanie pochodnych względem wag pierwszej warstwy

$$\frac{\partial E^n}{\partial w_{ji}^{(1)}} = \underbrace{\frac{\partial E^n}{\partial a_j^{(1)}}}_{\delta_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} \cdot z_i^{(0)}, \tag{11.3}$$

gdzie $\delta_j^{(1)}$ jest dalej obliczana jako pochodna złożona, uwzględniając fakt, że zmienna $a_j^{(2)}$ występuje w każdym składniku $a_k^{(2)}$ które miało połączenie z jednostką (neuronem) j warstwy pośredniej.
Aby obliczyć δ_j obliczmy ją jako pochodną złożoną, ponieważ E^n zależy od a_j za pośrednictwem a_k.

$$\delta_j^{(1)} = \left.\frac{\partial E^n}{\partial a_j^{(1)}}\right|_{a_j^{(1)}} = \sum_k \frac{\overbrace{\partial E^n}^{\delta_k^{(2)}} \partial a_k^{(2)}}{\cdot} \frac{\partial a_k^{(2)}}{\partial a_j^{(1)}}. \tag{11.4}$$

Należy teraz obliczyć $(\partial a_k^{(2)})/(\partial a_j^{(1)})$.
Przypomnijmy, że $a_k^{(2)} = \sum_j w_{kj} g(a_j^{(1)})$; wobec tego $(\partial a_k^{(2)})/(\partial a_j^{(1)}) = w_{kj}^{(2)} g'(a_j)$.
Tak więc

$$\boxed{\delta_j^{(1)} = \sum_k \delta_k^{(2)} w_{kj} g'(a_j) = g'(a_j) \sum_k \delta_k^{(2)} w_{kj}.} \tag{11.5}$$

Formuła ta określa **propagację wsteczną** błędów.
Aby obliczyć błąd δ_j dla jednostki j należy znać błędy δ_k warstwy nadrzędnej.

Wniosek końcowy:
Pochodne względem wag są funkcjami odpowiednich delt i aktywacji neuronów.

Spis literatury

[1] Ch. M. Bishop, *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1996.

[2] Ch. M. Bishop, *Neural networks: a pattern recognition perspective.* Technical Report NCRG/96/001, http://www.ncrg.aston.ac.uk/

[3] Hagan M.T., Demuth H.B., Beale M., *Neural Network design.* PWS Publishing Company, Thomson, Boston, 1996.

[4] Jolliffe I.T., *Principal Component Analysis*, Springer, New York 1986.

[5] Aleksander Michajlik, Witold Ramotowski, *Anatomia i fizjologia człowieka*, Wydawnictwa Lekarskie PZWL, Warszawa 1994.

[6] Morrison D.F. (1990). *Wielowymiarowa analiza danych statystycznych.* PWN Warszawa 1990. Tłum. z ang. *Multivariate Statistical Methods*, 2nd ed., McGraw-Hill 1981.

[7] T. Kohonen, *Self-organising Maps.* Springer, Berlin - Heidelberg, 1995.

[8] Józef Korbicz, Andrzej Obuchowicz, Dariusz Uciński, *Sztuczne sieci neuronowe. Podstawy i Zastosowania.* Akadem. Of. Wyd. PLJ, Warszawa 1994.

[9] Anna Kotula, *Sieci neuronowe i regresja na przykładzie pakietu Netlab.* Praca magisterska, Wrocław 2001.

[10] Ian Nabney, *Netlab: Algorithms for Pattern Recognition.* Springer 2001. Seria: Advances in Pattern Recognition. ISBN 1–85233–440–1.

[11] Netlab neural network software, Neural Computing Research Group, Division of Electric Engineering and Computer Science, Aston University, Birmingham UK, http://www.ncrg.aston.ac.uk/

[12] Stanisław Osowski, *Sieci neuronowe w ujęciu algorytmicznym.* WNT W-wa 1996.

[13] Sammon, J.W.Jr. (1969). A nonliner mapping for data structure analysis. *IEEE Trans. on Computers*, C-18(5), 401–409, May.

[14] Ewa Skubalska-Rafajłowicz, Samoorganizujące sieci neuronowe. W: M. Nałęcz (red), *Biocybernetyka i Inżynieria Biomedyczna 2000. Tom 6: Sieci neuronowe, str. 187–188*

[15] Ryszard Tadeusiewicz, *Sieci neuronowe*, Akademicka Oficyna Wydawnicza, Warszawa 1993.

[16] Tadeusiewicz R., *Elementarne wprowadzenie do techniki sieci neuronowych z przykładowymi programami.* Akademicka Oficyna Wydawnicza PLJ, Warszawa 1998.

[17] Johnson R.A., Wichern D.W. (1998). *Applied Multivariate Statistical Analysis.* Prentice Hall Inc., Englewood Cliffs, N.J. (4-th Edition).

[18] Juha Vesanto, SOM-based data visualization methods. *Intelligent Data Analysis*, 3 (2) 1999, 111–126. Rownież: Juha Vesanto, J. Himberg, E. Alhoniemi, J. Parhankangas, *SOM Toolbox for Matlab 5.* Som Toolbox team, Helsinki University of Technology, Finland, Libella Oy, Espoo 2000, 1–54. http://www.cis.hut.fi/projects/somtoolbox/

Alfabet Grecki

α	A		alfa
β	B		beta
γ	Γ		gamma
δ	Δ		delta
ϵ	E	ε	epsilon
ζ	Z		dzeta
η	H		eta
θ	Θ	ϑ	theta
ι	I		jota
κ	K		kappa
λ	Λ		lambda
μ	M		mi
ν	N		ni
ξ	Ξ		ksi
o	O		omikron
π	Π	ϖ	pi
ρ	P	ϱ	ro
σ	Σ	ς	sigma
τ	T		tau
υ	Υ		ypsilon
ϕ	Φ	φ	fi
χ	X		chi
ψ	Ψ		psi
ω	Ω		omega

Skorowidz